VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií

IFJ a IAL
2019/2020

Projekt - Implementace překladače imperativního jazyka IFJ19

Tým 007, varianta I

Kaláb Dominik	(25%)	xkalab11	vedoucí týmu
Lorenc Jan	(30%)	xloren15	
Staněk Vojtěch	(27%)	xstane45	
Pokorný Matěj	(18%)	xpokor78	

Brno, 30. Listopadu 2019

1. Úvod

Cílem projektu bylo navrhnout a implementovat interpret pro jazyk IFJ19, který je podmnožinou jazyka Python. Interpret je psán v jazyce C a v tomto dokumentu uvádíme postupy, kterými jsme se při jeho vytváření řídili.

2. Návrh a implementace

2.1. Datové struktury

Základním kamenem projektu jsou datové struktury pro rychlou a efektivní práci s daty v nich uložených. Námi implementované struktury jsou zásobník, binární vyhledávací strom, jednosměrně vázaný seznam a dynamický řetězec (jako v některých vyšších programovacích jazycích).

Binární vyhledávací strom slouží jako tabulka symbolů a je popsán dále.

Zásobník jsme potřebovali pro ukládání úrovně zanoření (indentace) a pro převod výrazu do postfixového zápisu.

Dynamický řetězec jsme implementovali hlavně kvůli tomu, abychom se vyhnuli nepříjemné práci s několikanásobnými ukazateli a omezenou pamětí při zpracovávání řetězce v jazyce C jako pole znaků.

Jednosměrně vázaný seznam jsme využili v podstatě jako dynamické pole pro ukládání výrazů a parametrů při volání funkce.

2.2. Tabulka symbolů

Tabulku symbolů jsme implementovali jako binární vyhledávací strom. Ta je nezbytná pro správný chod parseru a průběhu syntaktické analýzy. Do tabulky se ukládají identifikátory, které uchovávají informace o svém typu (proměnná a funkce) a další důležité informace (např. u funkcí počet parametrů, či zda je symbol definovaný). Nad tímto datovým typem jsme implementovali také několik užitečných funkcí. Vyhledávání ve stromě je prováděno voláním funkce searchST().

2.3. Lexikální analýza

Základním prvkem lexikální analýzy je deterministický konečný automat, bez jehož správného návrhu nemělo smysl vůbec s implementací začít.

Tu jsme se mírně odchýlili od běžných praktik učených na přednáškách, kde konečným stavem byl většinou platný stav, ze kterého již nešlo pokračovat. My jsme si místo toho určili

jeden konečný stav END, do kterého jsme v takovém případě přešli. Důvodem k tomuto návrhu je to, že nemůžeme ignorovat bílé znaky v počátečním stavu, neboť tam mohou vést na odřádkování, čili stavu INDENT. Tyto bílé znaky tedy ignorujeme v koncovém stavu END, kde nám výše zmíněný problém nehrozí, a potom, pokud v počátečním stavu narazíme na bílý znak, víme jistě, že máme odřádkovat. Z toho nám plyne, že koncovými stavy automatu jsou EOL, neboť kdybychom se znakem konce řádku šli do ENDu, tak vynecháme znaky odsazení, a dále stavy INDENT a END.

Implementace lexikálního analyzátoru spočívá v jedné ústřední funkci loadToken(), která načítá znaky a dle konečného automatu vrací aktuální token. Tato funkce je volána syntaktickou analýzou vždy, když potřebuje další token. Je tedy prováděna za běhu, nenaskenuje celý program hned na začátku a výsledek neposílá dál.

Přestože je analyzátor řízen konečným automatem, v jednom případě se odkloňuje. Ten nastává, pokud jsme v rámci odsazení poměrně zanořeni a vracíme se přes vícero úrovní. V takové situaci si zapamatuje, o kolik úrovní jsme se vrátili a při dalším volání funkce loadToken() vůbec nemusí dojít ke čtení znaků a průchodem konečným automatem, nýbrž rovnou ke generování tokenu dedent a to do té doby, dokud se úrovně nesrovnají.

2.4. Syntaktická analýza

Syntaktická analýza je definovaná pomocí LL-gramatiky (viz. příloha 5.2. a 5.3.) s použitím jednoho ε-pravidla. Samotná implementace je realizovaná metodou rekurzivního sestupu.

Jednotlivé neterminální symboly jsou zastoupené funkcí obsahující switch pro výběr následujícího pravidla na základě typu momentálně načteného tokenu. Možnost číst informace o momentálně zpracovávaném tokenu a načítání tokenů následujících nám umožňuje výše zmíněná lexikální analýza; její struktura Token_t a funkce loadToken(). Procházení programu pomocí syntaktické analýzy je hlavní proces interpretu, na základě kterého se volá lexikální analýza, probíhají sémantické kontroly a probíhá přímé generování kódu.

Další částí syntaktické analýzy je zpracování výrazů. Ty zpracováváme na základě precedenční tabulky (viz příloha 5.4). Tu jsme vytvořili na základě prezentací a přednášek IFJ, dále jsme čerpali v dokumentaci Pythonu.

Zpracováním výrazu se zabývá funkce expr(), která si dle potřeby volá další funkce. Funkce má dvě stěžejní části. První je převod výrazu do postfixové notace (kvůli lepšímu následnému zpracování a generování kódu), během které probíhá samotná syntaktická analýza. Druhou částí je sémantická analýza výrazu (funkce semantics()), která kontroluje typovou kompatibilitu operandů ve výrazu, a také zaručuje korektní přetypování v případě, že je to třeba.

2.5. Sémantická analýza

Sémantická analýza se zabývá převážně problémy definování proměnných a funkcí. K ukládání informací o funkcích a proměnných používáme dvě instance tabulky symbolů implementované pomocí binárního vyhledávacího stromu - tabulku lokální a tabulku globální.

Tabulku globální používáme k ukládání všech funkcí a k ukládání proměnných definovaných v hlavním těle programu. Tabulku lokální používáme k ukládání jednotlivých argumentů funkcí a proměnných definovaných v těle funkce. Dalším důležitým bodem v sémantické analýze je kontrola korektnosti volání funkce (počet argumentů). Sémantické kontroly jsou přímo zabudované do struktury procesu syntaktické analýzy implementované v parser.c.

Co se sémantické analýzy výrazů týče, její implementace nebyla jednoduchá, a to hlavně kvůli tomu, že IFJ19 je dynamicky typovaný jazyk. To znamená, že kontrolu kompatibility operandů ve výrazu je v mnoha případech možné provést až za běhu, neboť během analýzy ještě často nevíme, jaký typ budou proměnné zrovna bude mít.

2.6. Generování kódu

V tomto projektu jsme se rozhodli pro přímé generování kódu, tedy kód je generován již v průběhu syntaktické a sémantické analýzy. Tato volba nese z našeho pohledu výhody i nevýhody.

Mezi jednoznačné výhody patří jednoduchost implementace, není potřeba vytvářet žádný výstupní strom který by se následně předal modulu generátoru, s čímž souvisí i nižší nároky na výpočetní prostředky (paměť pro uložení stromu, procesorový čas pro vyhledávání a vkládání do stromu).

Mezi nevýhodami je nutné zmínit nutnost čekat s psaním generování kódu na dokončení syntaktického a sémantického analyzátoru, což právě v případě našeho projektu způsobilo, že jsme práci na něm dokončovali ještě pár hodin před pokusným odevzdáním. S větší provázaností se syntaktickou a sémantickou analýzou souvisí i nutnost v lepším případě ověřovat funkčnost generování, v horším přímo dělat změny v generování při každé změně v syntaktické a sémantické analýze. Jako poslední nevýhodu je nutné zmínit složitost předávání IFJcode19 identifikátorů proměnných a návěští v rámci programu v C, tuto lze ale elegantně řešit využitím zásobníkových instrukcí IFJcode19.

Z našeho pohledu tedy spíše nevýhody převažují nad výhodami, bohužel jsme k tomuto závěru došli až v průběhu, kdy už by bylo složité rozhodnutí tohoto rozměru měnit. V případě psaní překladače s optimalizacemi by těchto nevýhod pravděpodobně ještě přibylo.

Generování kódu probíhá v souboru parser.c, odkud se volají funkce ze souboru generator.c, které už přímo vypisují kód na stdout.

3. Vývojový cyklus a práce v týmu

Vývoj interpretu jsme rozdělili do 4 dvoutýdenních cyků. Na konci každého cyklu jsme měli schůzku, kde jsme popsali a zhodnotili odvedenou práci a naplánovali další běh.

1. Cyklus
   1.1. Datové struktury - Vojtěch Staněk
   1.2. Lexikální analýza - Jan Lorenc
2. Cyklus
   2.1. Syntaktická analýza hlavního těla programu - Dominik Kaláb
   2.2. Sémantická analýza hlavního těla programu - Dominik Kaláb
3. Cyklus
   3.1. Syntaktická analýza výrazů - Vojtěch Staněk
   3.2. Sémantická analýza výrazů - Vojtěch Staněk
4. Cyklus
   4.1. Generování struktury programu - Matěj Pokorný
   4.2. Generování funkcí a builtinů - Jan Lorenc

Poslední cyklus měl deadline nastavený týden před prvním pokusným odevzdáním, abychom měli čas na testování a opravy.

Převážná část komunikace probíhala přes prostředky Facebook Messenger, Microsoft Teams, Github a samozřejmě osobní setkání.

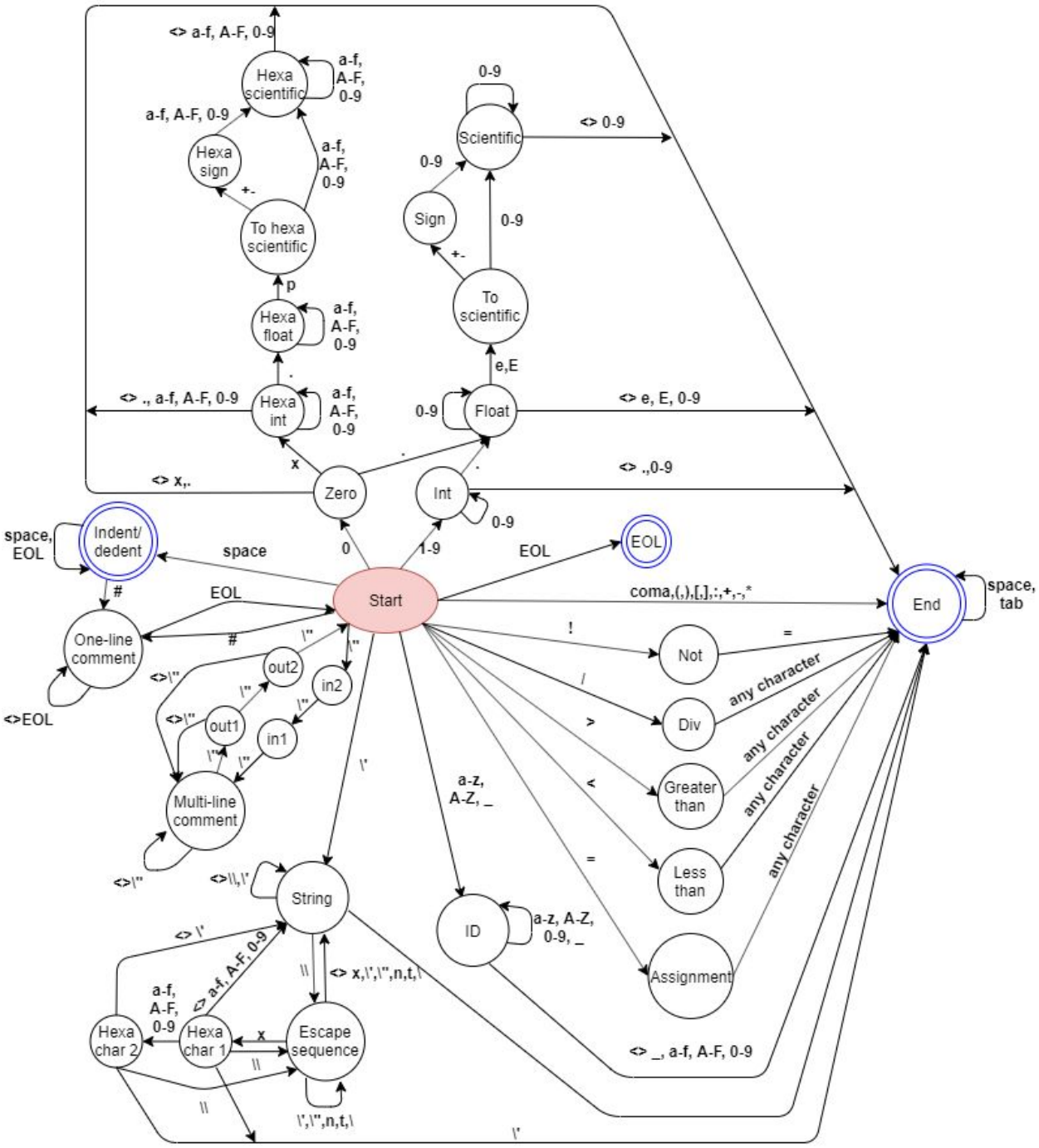
4. Závěr

S prací na projektu jsme relativně spokojení. Stihli jsme ho dokončit včas, přestože se v závěru u generování kódu naskytly nemalé problémy. S něčím takovým jsme však počítali, proto jsme si dopředu vymezili časovou rezervu. Ukázalo se tedy velmi dobré plánování vývoje a rozdělení práce.

Projekt nám bezesporu přinesl mnoho zkušeností a to nejen co se překladačů týče, ale i práce v týmu, projektového managementu, programování v jazyce C nebo používání gitu.

5. Přílohy

5.1. Konečný automat specifikující lexikální analyzátor



5.2. LL-gramatika

1	<prog>	->	<statement> <prog>
2	<prog>	->	EOF
3	<prog>	->	def id<params> : <eols> indent <statement> dedent <prog>
4	<params>	->	(<param>
5	<param>	->	<item> <param>
6	<param>	->	,<item> <param>
7	<param>	->	)
8	<statement>	->	while <expr> : <eols> indent <statement> dedent <statement>
9	<statement>	->	if <expr>: <eols> INDENT <statement> dedent else: EOL indent <statement> dedent <statement>
10	<statement>	->	id <assignOrFunc> <statement>
11	<statement>	->	<builtin> <params> <statement>
12	<statement>	->	return <expr> EOL <statement>
13	<statement>	->	pass EOL <statement>
14	<statement>	->	EOL <statement>
15	<statement>	->	ε
16	<assignOrFunc>	->	= <expression> <statement>
17	<assignOrFunc>	->	<params>
18	<assignOrFunc>	->	EOL
19	<item>	->	id
20	<item>	->	string
21	<item>	->	int
22	<item>	->	float
23	<builtin>	->	inputs
24	<builtin>	->	inputi
25	<builtin>	->	inputf
26	<builtin>	->	print
27	<builtin>	->	len
28	<builtin>	->	substr
29	<builtin>	->	ord
30	<builtin>	->	chr
31	<statement>	->	<expression>
32	<assignOrFunc>	->	<expression>
33	<eols>	->	EOL <eols>
34	<eols>	->	indent

5.3. LL-tabulka

α	id	string	int	float	=	def	EOL	EOF	indent	dedent	(	)	,	:	if	while	pass	return	inputs	inputi	inputf	print	len	substr	ord	chr
<prog>	1	1	1	1		3	1	2							1	1	1		1	1	1	1	1	1	1	1
<statement>	10	31	31	31			14								9	8	13	12	11	11	11	11	11	11	11	11
<params>											4															
<param>	5	5	5	5								7	6													
<assignOrFunc>					16		18				17															
<item>	19	20	21	22																						
<builtIn>																			23	24	25	26	27	28	29	30
<eols>							33		34																	

5.4. Precedenční tabulka

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