Projektová dokumentace

Implementace překladače imperativního jazyka IFJ20

2020/2021

Tým 008 - Varianta II

Martin Kneslík	**xknesl02**	25%
Adriana Jurkechová	xjurke02	25%
Karel Norek	xnorek01	25%
Petr Salaba	xsalab00	25%

Obsah

1 Úvod

Cílem tohoto projektu bylo vytvořit překladač imperativního jazyka IFJ20 (podmnožina jazyka Go). Překladač je napsán v jayzce C.

2 Implementace

Překladač má 3 hlavní části a to scanner, parser a generování kódu. Do scanneru patří lexikální analýza a do parseru syntaktická a sémentická analýza.

2.1 Lexikální analýza

Prvním krokem při tvorbě překladače byla implementace lexikální analýzy. Funkce **getNextToken** čte jednotlivé znaky ze zdrojového souboru a převadí je na strukturu **token**. Tato struktura obsahuje typ tokenu, který může být klíčové slovo, identifikátor, řetězec, celé nebo desetinné číslo, EOL, EOF, různé aritmetické a porovnávací znaky definované v jazyce IFJ20. Struktura taktéž obsahuje i atribut, který navazuje na typ tokenu. Atribut mají pouze tokeny typu řetězec, identifikátor, klíčové slovo, celé a desetinné číslo.

Lexikální analyzátor je vytvořený na základě námi vytvořeného konečného deterministického automatu. Tento konečný automat je implementovaný jako nekonečně se opakující switch, kde jednotlivé případy case zastupují jednotlivé stavy tohoto automatu. Postupně sa načítávají znaky, dokud se nevytvoří validní token, který vrátíme a ukončíme funkci. V případě, že znak na vstupu není povolený jazykem IFJ20, nastává lexikální chyba, kterou vrátime.

Při zpracování escape sekvencí využíváme pole, do kterého se postupně vkládají znaky, které můžou být v rozsahu 0-9, a-f nebo A-F. Toto hexadecimální číslo se převede na znak ASCII.

2.2 Syntaktická analýza

Syntaktická analýza je implementována metodou rekurzivního sestupu, kde je pro každé pravidlo v LL-gramatice vytvořena vlastní funkce. Všechny funkce pracují se strukturou **Parser**, ve které jsou všechny potřebné proměnné pro syntaktickou i sémantickou analýzu. Analýza začíná ve funkci **parse**, která je volaná z **main.c**.

Na začátku inicializuje danou strukturu **Parser** a zavolá první funkci z LL-gramatiky. Při implementaci používáme několik maker, které nám zjednodušují práci. Nejpoužívanější makro **getToken**, načte do proměnné ve struktuře token pomocí funkce **getNextToken** a zároveň zkontroluje návratovou hodnotu volané funkce. Pokud nastala jakákoliv chyba, tak hned vrátí chybový kód a analýza se přeruší. Dále pomocí proměnné **tokenProcessed** kontroluje, jestli je aktuální token již zpracován, pokud není, tak po zavolání makra zůstane aktuální token stejný. Nezpracovaný token, znamená to, že je momentálně epsilon pravidlo a načtený token už patří do jiného pravidla. Další makra **getType** a **getKeyword** načtou token a zkontrolují, jestli je token právě daný typ nebo klíčové slovo, jinak vrátí syntaktickou chybu a analýza skončí. Podobně funguje i makro **getRule**, které zavolá funkci zadaného pravidla a zkontroluje návratovou hodnotu.

2.2.1 Zpracování výrazů pomocí precedenční syntaktické analýzy

Speciálním případem je analýza výrazů, kde využíváme metodu syntaktické analýzy zdola nahoru.

K analýze výrazů jsme potřebovali sestavit precedenční tabulku. Operátory se stejnou asociativitou a prioritou jsme dali dohromady pro zjednodušení tabulky.

Po zavolání funkce **expression** ze samostatného souboru **expression.c** s hlavičkovým souborem **expression.h**. Použili jsme algoritmus probíraný na přednáškách IFJ. Začneme pushnutým $ na stack, značící začátek výrazu. V algoritmu používáme funkce **getIndexFromSymbol** a **getIndexFromToken** pro zjištění indexu v precedenční tabulce. Pro index = pushneme na stack) a zavoláme další token. Pro index < pushneme na stack pomocný znak **HANDLE**, pushneme symbol ze vstupu a zavoláme další token. Pro poslední index > kontrolujeme, jestli výraz, začínající znakem **HANDLE**, je platný podle daných pravidel. Pokud je platný, popneme tolik znaků, kolik je ve výrazu společně s HANDLE, poté pushneme na stack **NON_TERM** a zavoláme další token. Pokud výraz není platný, funkce vrací syntaktickou chybu.

Vykonáváme algoritmus dokud na vstup nepřijde znak ukončující výraz (značen $). Po vykonání zkontrolujeme, jestli se výraz podařilo zredukovat na pouze jeden **NON_TERM**. Pokud ne, vrátíme syntaktickou chybu.

Řešíme zde i rozlišení výrazu od funkce. Pokud je první token ve výrazu identifikátor, tak ho zkusíme najít v lokální tabulce symbolů. Pokud není identifikátor nalezen, tak se ukončí analýza výrazu a volání funkce se zpracuje normálně přes pravidla v LL-gramatice. Hned po ukončení tímto způsobem zkontrolujeme, zda je další token (. Pokud není, tak to znamená, že to není volání funkce, ale výraz, který obsahuje pouze nedefinovanou proměnnou - v tom případě nastane chyba číslo 3.

2.2.1.1 Stack pro precedenční analýzu

K precedenční syntaktické analýze využíváme stack pro syntaktickou analýzu, kde ukládáme symboly ze vstupu, popřípadě zjednodušený výraz podle pravidla.

Stack se nachází v souboru **stack.c** s hlavičkovým souborem **stack.h**. Funkce **stackInit** a **stackDispose** slouží k inicializaci a uvolnění stacku. Další dvojce funkcí je **stackPush** a **stackPop**. První zmíněná funkce pushne na stack zadaný prvním argumentem, hodnotu, zadanou druhým arugmentem. Druhá funkce popne poslední pushnutou hodnotu neboli vrchol stacku. Funkce **stackInserAfterTerm** vkládá pomocný znak **HANDLE** za první symbol co najde (bude se nacházet nad symbolem). Při hledání symbolu se ignoruje znak **NON_TERM**. Poslední funkcí je **stackTop**, která vrátí vrchol stacku.

2.3 Sémantická analýza

Během syntaktické analýzy kontrolujeme i různé sémantické chyby.

Tabulku symbolů jsme převzali z domácího úkolu do předmětu IAL a modifikovali ji pro naší potřebu. Tabulka pracuje se strukturou, která obsahuje proměnné potřebné pro sémantickou kontrolu funkcí a identifikátorů. Strukturu dat vrací všechny používané funkce a pracují s ní i makra, které jsme si pro ulehčení práce s tabulkou symbolů vytvořili. Tabulka globálních symbolů je uložena jako **sGlobal** ve struktuře **Parser**. Pro práci s lokální tabulkou symbolů jsme vytvořili zásobník, který pracuje právě se symtable. Operace push za zásobníku proběhne vždy na začátku rozsahu a pop na konci. Na vrcholu zásobníku je vždy lokální tabulka aktuálního rozsahu. Zásobník je uložen jako **sLocal** ve struktuře **Parser**.

Porovnávání datových typů u definice a přiřazování proměnných a volání/návratu z funkcí funguje na principu porovnávání dvou struktur typu string ze souboru **str.h**. Modul **str** jsme převzali ze stránek předmětu IFJ. Přidali jsme si vlastní funkce pro jednodušší vyrábění stringů.

Při kontrolování typů se porovnávají oba řetězce. V prvním řetězci jsou očekávané typy a ve druhém jsou typy, které musí odpovídat očekávaným typům. Řetězce se porovnávají po jednotlivých znacích. V řetězci se objevují znaky 0 až 4. 0-2 značí datový typ int, float64 a string. Znak 3 je používaný pro speciální proměnnou _, což znamená, že na daném místě může být jakýkoliv typ. Znak 4 se používá pouze v parametrech funkce **print**. Znamená to, že funkce může mít různé argumenty. Při kontrole typů se pak při tomto znaku celý proces kontroly přeskočí a vyhodnotí se, že typy sedí.

Při volání funkce, která není deklarovaná, se vytvoří záznam v globální tabulce symbolů. Uloží se argumenty, se kterými byla volána, očekávané návratové typy a označíme si, že funkce není deklarovaná. V následovné deklaraci funkce se kontroluje, jestli už byla funkce volána, a případně se porovná, jestli jsou datové typy v deklaraci a volání stejné. Na konci analýzy zpětně kontrolujeme, jestli byly všechny funkce deklarované. Kontrolujeme také, jestli se v globální tabulce symbolů nachází funkce **main** a jestli má správně definované argumenty a návratové hodnoty.

Kontrolu typů ve výrazech kontrolujeme tak, že po prvním nálezu termu nebo identifikátoru si poznačíme daný datový typ a pak už jenom kontrolujeme, zda je další načtený datový typ stejný. Poznačíme si, zda je ve výrazu porovnávání, a po výrazu zkontrolujeme, zda bylo porovnávání očekáváno.

2.4 Generování kódu

Pro generování kódu jsme zvolili metodu generování instrukcí tříadresného kódu za běhu parsování, a poté překlad výsledné sekvence instrukcí na výstupní kód.

Během parsování se v určitých místech volají funkce generátoru, který podle kontextu převzatého z parseru vytvoří patřičný tříadresný kód nebo případně manipuluje se zásobníkem symbolů. Symbol je struktura která se skládá z obsahu a typu symbolu, který je využíván jako jednotlivé adresy tříadresného kódu. Sekvence tříadresných instrukcí je implementován jako spojový seznam bloků, které obsahují statické pole ukazující na 64 instrukcí, za účelem zvýšení rychlosti alokace a vyhledávání. Zmíněná struktura také obsahuje mezipaměť, která si pamatuje poslední přistoupený blok pro rychlejší sekvenční čtení či zápis.

Názvy proměnných a funkcí se generují dynamicky podle adres objektů a případně aktuálního rámce, čímž se vyřeší případný problém extrémně dlouhých názvů proměnných, a má to i za vedlejší účinek obfuskaci názvů identifikátorů.

Při generování tříadresného kódu jsme museli kontrolovat, pokud definujeme proměnnou uvnitř cyklu, kde musíme dosáhnout toho, abychom se nedostali do situace definování proměnné, která už existuje. Jako řešení na začátek každé funkce vytváříme odkaz na separátní sekvence tříadresných instrukcí, které se ukládají ve spojovém seznamu, a slouží pro zápis všech definicí proměnných, které by jinak způsobily pád programu.

Vestavěné funkce jsme řešili tak, že je máme uložené ve struktuře jako řetězec, a pokud detekujeme při generaci kódu, že je daná vestavěná funkce použita, tak ji vložíme na začátek vygenerovaného programu. Funkci print, která má nedefinovaný počet argumentů řešíme separátně, kde tříadresný kód pro výpis symbolů se generuje přímo, bez volání žádných funkcí pomocí instrukce **CALL**.

Po procesu generování tříadresného kódu započne generování výstupního kódu, kde sekvenčně procházíme vygenerovaný seznam tříadresných instrukcí a pro každou instrukci nezá-

visle na kontextu vygenerujeme na standardní výstup sekvenci instrukcí ve výstupním jazyce IFJcode20.

3 Práce v týmu

3.1 Komunikace

Kvůli koroně jsme se nemohli setkat osobně, proto jsme se rozhodli založit discord server pro náš tým. Na discordu jsme měli pravidelné schůzky, kde jsme probrali jak na tom kdo je a co je potřeba vyřešit. Využívali jsme i možnost streamovat přes Discord, když byla nějaká nejasnost v kódu.

3.2 Verzovácí systém

Jako správu verzí jsme použili verzovací systém Git. Pro vzdálený repozitář jsme používali GitHub.

3.3 Rozdělení práce

Práci na projektu jsme si na začatku rozdělili rovnoměrně podle obtížnosti a časové náročnosti. Shodli jsme se, že každý by měl dostat 25%, protože odvedl svou zadanou práci.

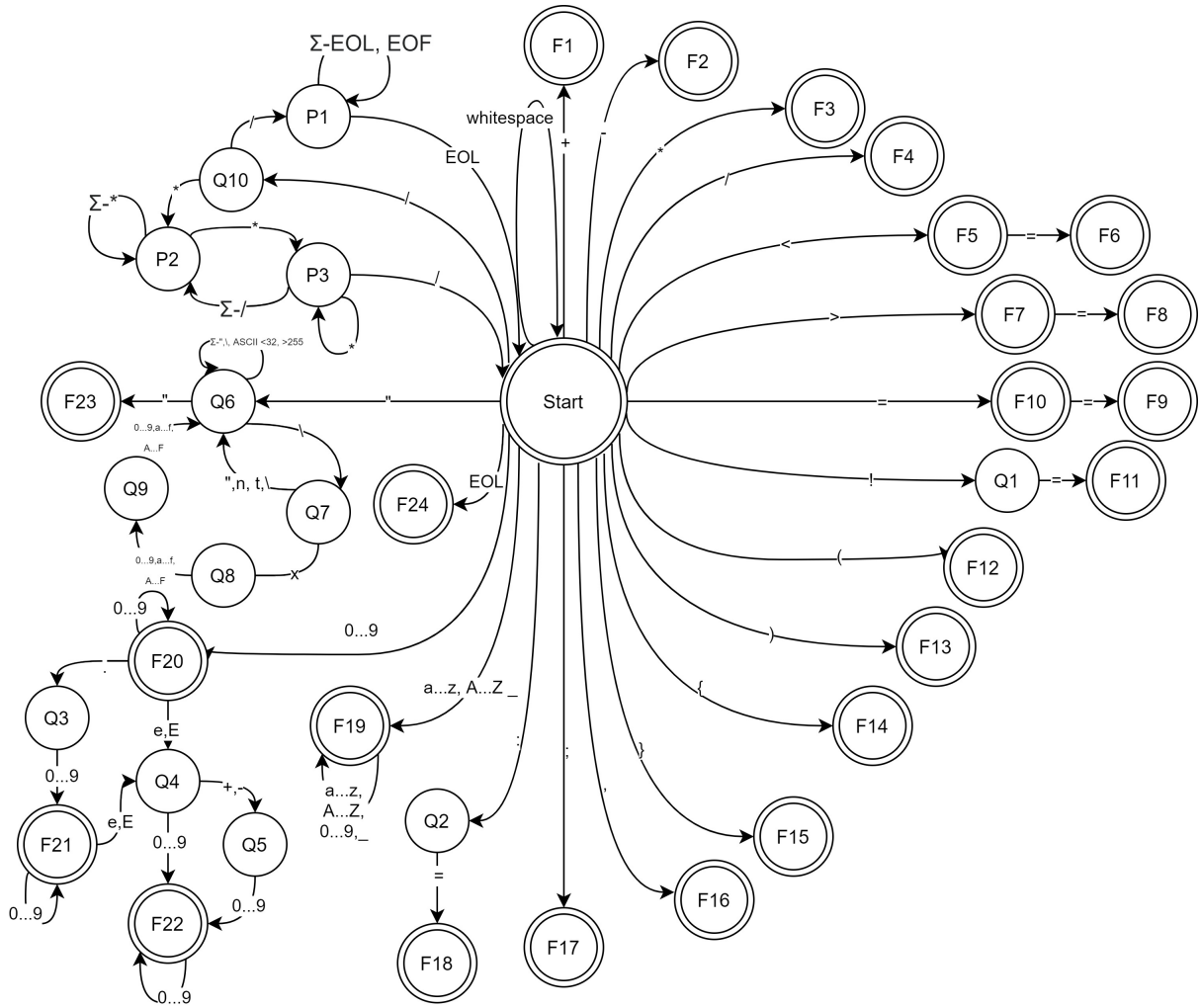
Vetšina úkolů byla rozdělena tak, že jeden člen týmu na něm pracoval a další člen mu pomáhal pokud se někde zasekl a potřeboval poradit.

Martin Kneslík	vedení týmu, syntaktická analýza, symtable, sémentická analýza
Adriana Jurkechová	lexikální analýza, konečný automat pro lexikální analýzu
Karel Norek	testování, precedenční analýza výrazů, stack, dokumentace
Petr Salaba	správa git repozitáře, generování mezikódu, vestavěné funkce, Makefile

Tabulka 1: Rozdělení práce

Každý člen týmu napsal dokumentaci pro část, kterou implementoval.

A Konečný automat pro lexikální analýzu



F1 PLUS
F2 MINUS
F3 MULTIPLICATION
F4 DIVISION
F5 LESS THAN
F6 LESS OR EQUAL
F7 MORE THAN
F8 MORE OR EQUAL
F9 EQUAL
F10 ASSIGN
F11 NOT EQUAL
F12 LEFT BRACKET
F13 RIGHT BRACKET
F14 LEFT CURLY BRACKET
F15 RIGHT CURLY BRACKET
F16 COMMA
F17 SEMICOLON
F18 VARIABLE DEFINITION
F19 IDENTIFIER
F20 INTEGER
F21 DECIMAL
F22 NUMBER WITH EXPONENT
F23 STRING
F24 EOL

S START
Q1 NOT EQUAL
Q2 VARIABLE_DEF
Q3 DECIMAL_POINT
Q4 SCANNER_STATE_EXPONENT
Q5 SCANNER_STATE_EXPONENT_SIGNED
Q6 SCANNER_STATE_STRING
Q7 SCANNER_STATE_ESCAPE
Q8 SCANNER_STATE_CHARACTER
Q9 SCANNER_STATE_SECOND_CHARACTER
Q10 SCANNER_STATE_COMMENTARY
P1 COMMENTARY
P2 SCANNER_STATE_BLOCK_COMMENTARY_START
P3 SCANNER_STATE_BLOCK_COMMENTARY_EXIT

B LL - gramatika

1. <package> -> PACKAGE ID EOL <prog>
2. <prog> -> FUNC ID (<params>) <ret> { EOL <body> } EOL <prog>
3. <prog> -> EOL <prog>
4. <prog> -> EOF
5. <params> -> ID <type> <params_n>
6. <params> -> ε
7. <params_n> , ID <type> <params_n>
8. <params_n> -> ε
9. <ret> -> (<ret_params>)
10. <ret> -> ε
11. <ret_params> -> <type> <ret_params_n>
12. <ret_params> -> ε
13. <ret_params_n> -> , <type> <ret_params_n>
14. <ret_params_n> -> ε
15. <type> -> INT
16. <type> -> FLOAT64
17. <type> -> STRING
18. <body> -> FOR <for_definition> ; <expression> ; <for_assign> { EOL <body> } EOL <body>
19. <body> -> IF <expression> { EOL <body> } ELSE { EOL <body> } EOL <body>
20. <body> -> RETURN <ret_values> EOL <body>
21. <body> -> ID <body_n> EOL <body>
22. <body> -> EOL <body>
23. <body> -> ε
24. <body_n> -> <definition>
25. <body_n> -> <assign>
26. <body_n> -> <func>
27. <id_n> -> , ID <id_n>
28. <id_n> -> ε
29. <for_definition> -> ID <definition>
30. <for_definition> -> ε
31. <for_assign> -> ID <assign>
32. <for_assign> -> ε
33. <value> -> ID <func>
34. <value> -> <expression> <expression_n>
35. <expression_n> -> , <expression> <expression_n>
36. <expression_n> -> ε
37. <definition> -> := <expression>
38. <assign> -> <id_n> = <value>
39. <func> -> (<arg>)
40. <arg> -> <term> <term_n>
41. <arg> -> ε
42. <term> -> ID
43. <term> -> VALUE_INT
44. <term> -> VALUE_FLOAT64
45. <term> -> VALUE_STRING
46. <term_n> -> , <term> <term_n>
47. <term_n> -> ε
48. <ret_values> -> <value>
49. <ret_values> -> ε

C LL - tabulka

	PACKAGE	ID	FUNC	(	)	{	}	,	;	INT	FLOAT64	STRING	FOR	IF	RETURN	:=	=	VAL_INT	VAL_FLOAT64	VAL_STRING	EOL	EOF
<package>	1																					
<prog>			2																		3	4
<params>		5			6																	
<params_n>					8			7														
<ret>				9		10																
<ret_params>					12					11	11	11										
<ret_params_n>					14			13														
<type>										15	16	17										
<body>		21					23						18	19	20						22	
<body_n>				26				25								24	25					
<id_n>								27									28					
<for_definition>		29							30													
<for_assign>		31				32																
<value>		33																				
<expression_n>								35													36	
<definition>																37						
<assign>								38									38					
<func>				39																		
<arg>		40			41													40	40	40		
<term>		42																43	44	45		
<term_n>					47			46														
<ret_values>		48																			49	

D Precedenční tabulka

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