Dokumentácia projetku IFJ

Implementácia prekladača pre jazyk IFJ19

Tím 054, varianta 2

Peter Vinarčík – xvinar00

Lukáš Javorský – xjavor20

Marián Lorinc – xlorin01

Patrik Ondriga – xondri08

9.12.2019

Obsah

1 Úvod

Dokumentácia slúži na popis návrhu a implementácie imperativného jazyka IFJ19 v jazyku C. Program načítava kód v tomto jazyku a preloží ho do cieľového jazyka IFJcode19.

Projekt je určený pre predmety IFJ (Formálne jazyky a prekladače) a IAL (Algoritmy) na FIT VUT

2 Tímová spolupráca

2.1 Verzovací systém

Pri implementácii projektu sme ako verzovací systém využili git. Ako server našeho git repozitára sme zvolili GitHub. Pri problémoch, bugoch, alebo nejasnostiach sme vytvárali relevantné issues, ktoré boli ďalej riešené. Pokiaľ bola issue vyriešená, tagla sa v korešpondovanom Pull-Requeste.

Pre jednotlivé features programu sme si vytvorili vlastné branche a mergovanie do mastra bolo kontrolované pomocou Pull-Requestov, ktoré museli aspoň dvaja členovia tímu odsúhlasiť. Pokiaľ na jednotlivej feature spolupracovalo viacej členov, každý z nich si vytvoril privátnu branch s názvom odpovedajúcej featury a s prefixom “private-login-“.

2.2 Komunikácia

Komunikačný kanál v tíme bol primárne Discord. Väčšinu implementačných vecí sme riešili tam v špecifickej miestnosti. Pri menej dôležitých veciach sme využívali Messanger.

Taktiež sme sa pravidelne stretávali a predávali si informácie o tom, na čom sa práve pracuje, a či nepotrebuje niekto s niečím pomôcť/poradiť.

3 Implementácia

3.1 Lexikálna analýza (Skener)

Lexikálna analýza je prevádzaná pomocou konečného automatu skeneru (viz. Príloha 1), ktorý sa vo väčšine skladal z C príkazu switch. Lexikálna analýza indentácie (Indent, Dedent) nie je súčasťou automatu, pretože kontrolovanie odriadkovania je pred každým volaním funkcie get_token(). Indentácia bola riešená pomocou zásobníku, ktorý mal natvrdo zadanú nulu na spodku zásobníka. Pokiaľ prišlo viacej medzier pred začiatkom príkazu, ich počet sa pridal na vrch zásobníka. Pokiaľ bol počet medzie väčší ako vrch zásobníka, odošle sa token INDENT. Pokiaľ je počet medzie menší ako vrch zásobníka, tak sa rekurzívne popuje až kým bude ich počet rovný alebo väčší, a za každým popom sa posiela token DEDENT.

3.2 Syntaktická analýza (Parser)

Syntaktická analýza je srdcom našeho programu. Vyhodnocuje všetky chyby, podľa stanovených pravidiel. Je implementovaný pomocou metódy rekurzívneho zostupu, založeného na LL gramatike (Príloha 2), okrem spracovania výrazov. Spracovanie výrazov sa robí pomocou precedenčnej tabulky (Príloha 3). Parser volá funkciu skeneru get_token a pomocou nech sa dalej rozhoduje čo robiť. Parser je rozdelení na dve časti. Jedna je pre výrazy a druhá pre všetko ostatné. Parser pre výrazy je lokalizovaný v súboroch expression.*. Druhá a väčšinová časť je lokazivoná v parser.*. Parser pre výrazy je volaný hlavným parserom.

3.3 Sémantická analýza

Sémantická analýza zisťuje, či je možné robiť už aj lexikálne aj syntakticky správne operácie v kóde. Inak povedané, či robí užívateľ naozaj to čo chcel/môže. Sémantická analýza je riešená dvoma spôsobmi. Prví z nich je ešte pred generovaním kódu a to tak, že skontroluje názvy funkcii, či už náhodou s takým identifikátorom neexistuje, či sa nepoužíva premenná, ktorá ešte zadeklarovaná nebola. Druhý je typová kontrola vo výrazoch. Táto kontrola je až po generovaní kódu a analyzuje, či náhodou užívateľ nevyužíva operácie nad nekompatibilnými typmi. Napríklad či nechce pripočítať celočíselnú premennú s premennou typu string.

4 Generovanie kódu

Celý vstupný program sa uloží do jedného veľkého ast stromu. Tento strom sa ďalej spracúva a generátoru sú posielané samotné podstromy. Podľa typu podstromu sa určí ktorý kód sa bude generovať do tela funkcie $$main. Pri volaní generovania podstromu je zaručené, aby sa tam nenachádzali žiadne lexikálne,syntaktické a sémantické chyby. Jedninou výnimkou sú výrazy. Pri nich sa sémantická kontrola vykonáva až v samotnom vygenerovanom kóde. Toto riešenie sme zvolili, aby užívateľ mohol používať vstavané funkcie a mal zaručené, že mu jeho program nespadne.

5 Pomocné štruktúry

5.1 Stack

Jedna z najhlavnejších pomocných štruktúr je zásobník (ang. Stack). Implementáciu zásobníka z predmetu IAL sme upravili a prispôsbili na náš projekt. Taktiež funkcie, ktoré sú pre zásobníkovú štruktúru boli upravené, aby boli použiteľné pre náš projekt. Súbory obsahujúce túto štruktúru sme nazvali stack.*.

5.2 Queue

Ďalšiu z pomocných štruktúr, ktorú sme použili je fronta (ang. Queue). Implementáciu fronty sme znova čerpali z predmetu IAL a následne sme ju upravili a prispôsobili pre náš kód. Frontu sme implementovali pomocou dequeue čo je obojstranná queue. Dequeue sme zvolili pre prípadnú núdzu práve takejto štruktúry. Queue je následne ukrátená o funkcie oboch smerov. Súbory, ktoré špecifikujú frontu sú nazvané queue.*.

5.3 Symtable

Symtable je implementácia hashovacej tabuľky. Názov symtable bol povinný a preto bude ďalej spomínané len toto meno. Symtable bola povinnostou pre variantu 2, ktorej ako tím sme. Jej využitie bolo hlavne v sématickej analýze, pretože sú v nej uložené premenné či už lokálnej alebo globálnej scopy. Taktiež sa využíva pri vytváraní nových premenných alebo funkcii, aby nedošlo k tomu, že bude viacej premenných/funkcii s rovnakým identifikátorom. Ďalej sa využíva pri zisťovaní, či premenná, ktorá je súčasťou výrazu už bola zadeklarovaná alebo nie. Súbory, ktoré priliehajú symtable sme nazvali symtable.*.

6 Pomocná knižnica String

V našom programe sme začali implementáciou pomocnej knižnice ptr_string. Táto funkcia slúži pre zjednodušenie našej práce s vlastnou štruktúrou ptr_string_t, ktorú využívame vo veľmi veľa prípadoch. Jedným z týchto prípadov je aj lexikálna analýza. Token akožto návratová hodnota skeneru, má štruktúru obsahujúcu value a token_type. Value je typu ptr_string_t, a je to vlastne mutable string, ktorý vďaka pomocnej knižnice nie je zložitý na prácu.

7 Rozšírenia

7.1 CYCLES

Naša implementácia podporuje štruktúry continue, break a while-else. Štruktúra for nie je podporovaná. Pri implementácií whilu sme využili, že parsujeme kód do AST a následne vytvárame scope pre štruktúry ako sú napríklad funcdef, while, if. Scope obsahuje referencie na tieto štruktúry, ktoré obsahujú číslo riadku. V prípade continue sa volá inštrukcia jump na while$riadok label, v prípade break skočí na ENDWHILE$riadok. Else klauzula vo while sa vloží do stacku generovania a použije sa pomocný scope pre generovanie labelov.

7.2 IFTHEN

Podmienky sú implementované tak, že if, elif, else podstromy AST obsahujú hranu do podstromu alternate, kde je uložená ďalšia podmienka. Pri else klauzule je hrana na

alternate NULL. Podmienky sa vyhodnocujú na začiatku. Ak je nejaká podmienka pravdivá zavolá sa inštrukcia jump na label if$line, elif$line, else$line. Ak ani jedna nie je pravdivá, zavolá sa inštrukcia jump na label else$line alebo endif$line.

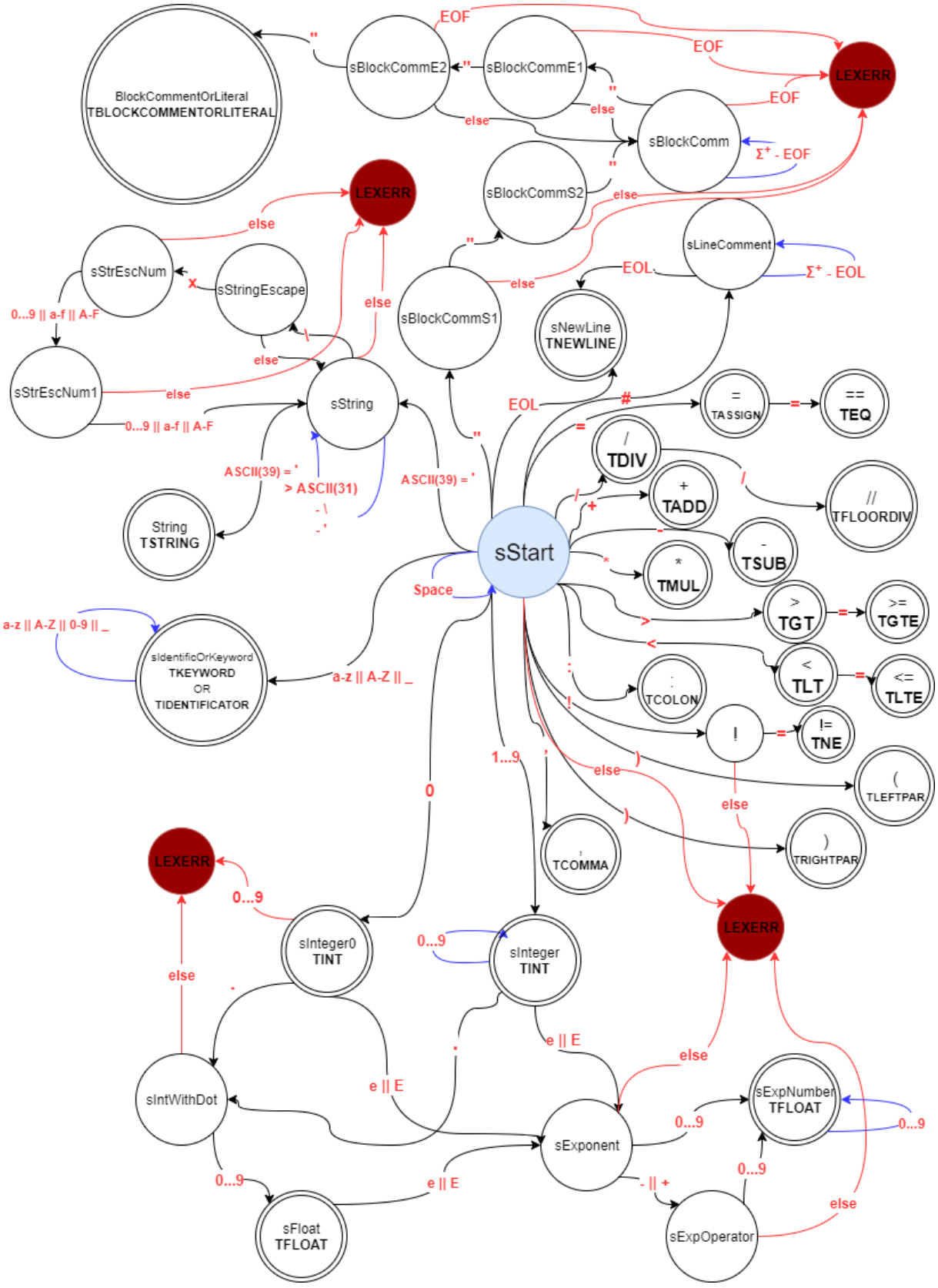
8 Záver

Projekt nás naučil spolupracovať ako tím. Veľa sme sa popri ňom naučili v pravidelnej komunikácii.

Ďalšie zo zistení pri takomto veľkom projekte bolo, že treba projekt robiť v predstihu, aby sme sa vyvarovali nevyspytateľným chybám na konci odovzdania.

9 Prílohy

9.1 Automat



9.2 LL Gramatika

<program> => <program_definiton> <statement_list> <empty_lines>
<program_definition> => ε
<program_definition> => <function_call> <function_definition> <variable_definition_list> <empty_lines>
<program_definiton>
<function_defition> => <function_name> INDENT <variable_definition_list> <statement_list> <function_call>
<empty_lines> DEDENT
<id_list> => <id> <const>
<id> => ID
<const> => NUMBER
<function_name> => DEF <id>(<param_list>): EOL
<param_list> => <param> <param_list>
<param> => <id>
<variable_definition> ε
<variable_definition> <id> <intialization> EOL
<variable_definition_list> <variable_definition> <variable_definition_list>
<function_call> => ε
<function_call> => <id>(<param_list>) EOL
<empty_lines> => EOL <empty_lines>
<empty_lines> => ε
<initialization> => = <expression>
<statement_list> => <statement> <statement_list>
<statement_list> => ε
<statement> => <if_statement> <while_statement>
<statement> => RETURN <expression>
<statement> => <id> = <expression> EOL
<if_statement> => IF <expression_compare_list> : EOL INDENT <variable definition_list> <statement_list>
<empty lines> <expression> DEDENT
<expression> => <expression_list>
<expression_list> => <expression_add> <expression_sub> <expression_mul> <expression_div>
<expression_floordiv> <expression_list>
<expression_add> => <id_list> + <id_list>
<expression_add> => ε
<expression_sub> => <id_list> - <id_list>
<expression_sub> => ε
<expression_mul> => <id_list> * <id_list>
<expression_mul> => ε
<expression_div> => <id_list> / <id_list>
<expression_div> => ε
<expression_floordiv> => <id_list> // <id_list>
<expression_floordiv> => ε
<expression_compare_list> => <expression_gt> <expression_gte> <expression_lt> <expression_lte>
<expression_neq> <expression_eq>
<expression_gt> => <id_list> < <id_list>
<expression_gt> => ε
<expression_gte> => <id_list> <= <id_list>
<expression_gte> => ε
<expression_lt> => <id_list> < <id_list>
<expression_lt> => ε
<expression_lte> => <id_list> <= <id_list>
<expression_lte> => ε
<expression_neq> => <id_list> != <id_list>
<expression_neq> => ε
<expression_eq> => <id_list> == <id_list>
<expression_eq> => ε

9.3 Precedenčná tabuľka

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