Fakulta informačních technologií
Vysoké učení technické v Brně

Formální jazyky a překladače

Dokumentace projektu

Tým 028 – varianta II

Jan Klhůfek – vedoucí (xklhuf01, 34%)
Andrea Chimenti (xchime00, 33%)
Martin Šerý (xserym01, 33%)
Matej Alexej Helc (xhelcm00, 0%)

11. prosince 2019

1 Úvod

Naším úkolem bylo vytvořit překladač pro jazyk IFJ19. Úkolem překladače je načíst vstup ze standardního vstupu a na standardní výstup vytisknout kód v jazyce IFJcode19. Jazyk IFJ19 je podmnožinou jazyka Python, jazyk IFJcode19 je jazyk který připomíná assembler, ale zjednodušuje mnoho úkonů.

2 Lexikální analýza – scanner

Lexikální analýzu jsme založili na deterministickém konečném automatu (DKA) doplněném o pomocný zásobník. O zpracování většiny příchozích lexému ze standardního vstupu se stará samotný automat, kde jsou jednotlivé stavy reprezentovány prvky v poli `enum`ů. Speciálně je pak přistupováno ke generování tokenů INDENTU a DEDENTU, které se pro větší přehlednost zpracovávají mimo automat.
Implementace automatu je pak založena na `switch`i stavů automatu, který postupně načítá znaky ze vstupu a v závislosti na typu znaku se rozhodne, do jakého stavu přejít. Následně načítá další znaky dokud nedojde do koncového stavu a tím vygeneruje příslušný token předávaný do parseru.
Tokeny INDENT a DEDENT se mohou vyskytovat pouze na začátku řádku a tak se zpracovávají samostatně mimo automat. Tím jsme minimalizovali riziko vzniku nesprávného zacházení s mezerami uvnitř automatu a zpřehlednili tím i samotný automat.
V případě, že do nekoncového stavu přijde neočekávaný znak, lexikální analýza je neúspěšná. Z lexikálního analyzátoru se do parseru ve struktuře předává načtený token spolu s návratovou hodnotou značící úspěch, potažmo neúspěch lexikální analýzy.

3 Syntaktická analýza – parser

3.1 Analýza shora dolů

Pro syntaktickou analýzu shora dolů jsme zvolili rekurzivní sestup založený na LL gramatice a LL tabulce. Pro každé pravidlo v LL gramatice je vytvořená funkce, která simuluje rozderivování daného pravidla voláním jiných takovýchto funkcí. Pro spuštění syntaktické analýzy se spustí funkce `prog`, která reprezentuje počáteční neterminál. Veškerá data jsou předávana přes stukturu `tParser_data`, která obsahuje odkaz na tabulku symbolů, zásobník, informace o tokenech, různé flagy a jiné pomocné struktury. Alokace těchto dat se vykoná vždy před samotným spuštěním analýzy a dealokace je provedena vždy před ukončením programu. Nemělo by tedy hrozit, že by program obsahoval úniky paměti (pokud nedojde k chybě typu `segmentation fault`, kterou se nám v žádných z našich testů nepodařilo vyvolat).

3.2 Analýza zdola nahoru

Zpracování výrazů je založeno na precedenční tabulce a zásobníku symbolů. V kódu je tabulka reprezentována 2D polem znaků, kde indexy řádků značí symbol na zásobníku a indexy sloupců načtený token ze vstupu. Znak na průniku indexů v tabulce určuje precedenční prioritu symbolu na zásobníku vůči načtenému tokenu.
Postupně se vždy nejprve zjistí aktuální symbol na vrcholu zásobníku a načtený token ze vstupu a podle tabulky se provede zpracování. Buď se token vloží na zásobník, generuje se `handle` pro budoucí derivaci některého pravidla nebo se provede samotná derivace pravidla.
V rámci derivace se taktéž provádí sémantické kontroly pro přetypování, běhové chyby a volá se generování cílového kodu. Při syntaktické analýze výrazů se zároveň pro každý neterminál vytvoří uzel do dílčí tabulky symbolů, používané pouze pro právě zpracovávaný výraz, a na konci precedenční analýzy se průchodem vytvořeného abstraktního syntaktického stromu generuje postfixový tvar výrazu, který se posílá generátoru pro zpracování.
Snahou bylo odhalit co nejvíce možných chyb a neplatných konstrukcí již během překladu, abychom si tak ulehčili práci s generováním kódu.

4 Sémantická analýza

Sémantická analýza je implementována přímo do kódu parseru. Při definování funkce a vytvoření nové proměnné se do tabulky symbolů uloží záznam s funkčními daty pro daný typ. Při jakémkoliv jiném přístupu k identifikátorům, je nejprve zkontrolováno, zda identifikátor již byl definován a pokud ne, je vyvolaná příslušná chyba.

5 Generování cílového kódu

Generátor cílového kódu obsahuje metody, které tisknou na standardní výstup instrukce v jazyce IFJ19. Tyto metody jsou volány výrazy a parserem. Některé metody přijimají parametry určující například jména proměnných nebo návěští. Generátor obsahuje vestavěné funkce, které se vygenerují hned při startu generování.

6 Tabulka symbolů

Implementace tabulky symbolů je založena na hashovacích tabulkách. Samotná tabulka symbolů (struktura `tSymtable`) je zásobník odkazů na hashovací tabulky. Záznamy definovaných proměnných se ukládájí vždy do hash tabulky na vrchol zásobníku a vyhledávání záznamů probíhá postupně také od nejvyšší. V případě, že je ze zásobniku tabulka vyjmuta, jsou záznamy proměnných v dané tabulce odstraněny a uvolněny z paměti.

7 Práce v týmu

V počátcích se práce v týmu zdála být bezproblémová, organizovali jsme pravidelné srazy, kde jsme si přerozdělili práci a domluvili se, že na každou část by měli být přiděleni dva členové týmu, aby si mohli navzájem radit a pomoci si. Taktéž jsme tím chtěli zajistit určitou míru refaktorizace kódu. Bohužel s přibývajícími projekty a půlsemestrálními zkouškami bylo méně času, a tak docházelo k odkladům a menšímu důrazu na dodržení deadlinů. To se nejvíce podepsalo na čtvrtém členu týmu, který navzdory častým příslibům dodání potřebných pomocných knihoven, nebyl schopen práci dokončit. Postupně se načítali výmluvy, že se nemůže dostavit na sraz nebo že kód dodá do pár dnů, i když to byla otázka jednoho odpoledne. Pár dnů před pokusným odevzdáním se totiž ukázalo, že mu programování dělá potíže a nesvěřil se nám s tím už na začátku. Po vzájemné domluvě ohledně přerozdělování bodů padl z jeho strany návrh tým zcela opustit, jelikož je to pro něj veliká výzva a nezvládá ji. Tak jsme byli nuceni práci dokončit ve třech. Na jednu stranu nám přibylo práce s předěláváním částí po odešedším členu, ale těch naštěstí nebylo mnoho. Navzdory očekávání nám komunikace lépe fungovala a byli jsme schopni projekt téměř zcela dokončit. V posledních fázích jsme často provozovali metodu pair-programmingu, která nám pomohla k odhalení a opravení mnohých nedokonalostí.

7.1 Komunikace

Komunikace probíhala hlavně verbálně a to buď osobně, nebo prostřednictvím hovorů přes Skype. Během semestru jsme pořádali schůzky, kterých se účastnili všichni aktivní členové týmu. Na schůzkách byla vždy rozdělena práce a zadány nové úkoly.

7.2 Verzovací systém a použité nástroje

Pro verzování a zálohování kódu byla zvolen git s hostingem na GitLab. Pro každý logický celek byla vytvořena vlastní větev. Po dokonční práce se změny spojovaly do hlavní vývojové větve. Ve finálních fázích vývoje bylo využito možnosti live sharingu editorů Visual Studio a Visual Studio Code.

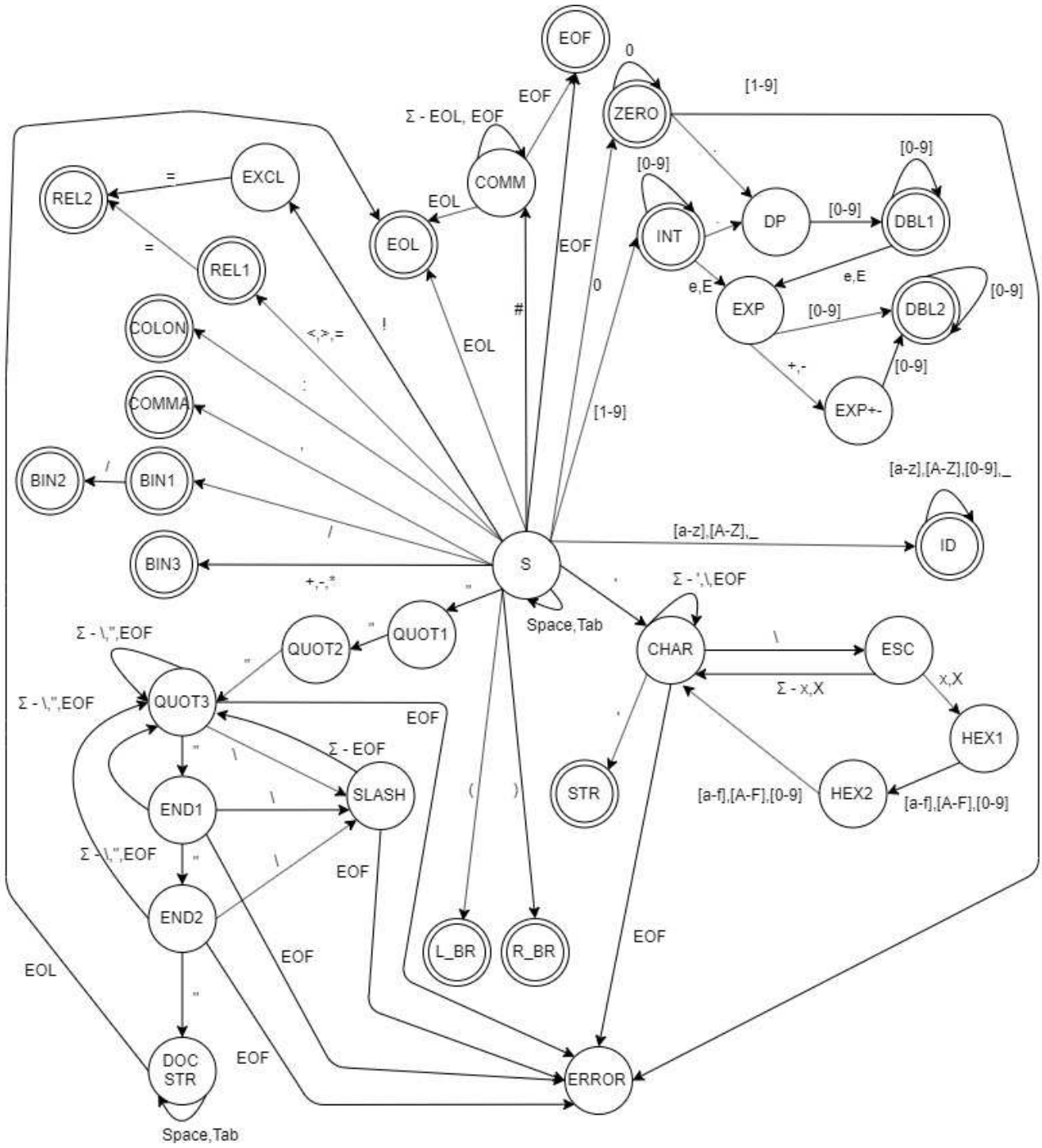
7.3 Rozdělení práce

- **Jan Klhůfek:** lexikální analýza, precedenční analýza, sémantická analýza, vedení týmu.
- **Andrea Chimenti:** syntaktická analýza shora dolů, sémantická analýza.
- **Martin Šerý:** generování cílového kódu, dokumentace, testování.
- **Matej Alexej Helc:** člen se na vývoji nepodílel a tým opustil.

8 Neimplementované části

Vzhledem k odchodu člena z týmu, nebyl bohužel projekt zcela dotažen do konce. Následující části nejsou z časových důvodu implementovány:

- **Definování vlastních funkcí:** Překladač umí definice funkcí lexikálně a syntakticky zkontrolovat. Ke kódu uvnitř funkce se však chová jako ke kódu v hlavním těle programu. Při pokusu o zavolání vlastní funkce, vyhodí překladač sémantickou chybu č. 3.
- **Funkce `print` a `substr`:** Funkce `print` umožňuje vytištění pouze jednoho parametru. Funkce `substr` není implementována a při volání dojde k sémantické chybě č. 3.
- **Definování proměnné v cyklu while:** V příkazu cyklu `while` není možná prvotní definici proměnné. Interpret zahlásí chybu č. 52 z důvodu již definované proměnné při vícenásobném procházení cyklem.



Obrázek 1: Konečný deterministický stavový automat

```
(1)  PROG → SEQUENCE PROG
(2)  PROG → "def" "id" "(" PARAMS ")" ":" "EOL" "INDENT" STATEMENTS "DEDENT" PROG
(3)  PROG → "EOF"
(4)  PROG → "EOL" PROG
(5)  PARAMS → "id" PARAMS_N
(6)  PARAMS → ε
(7)  PARAMS_N → "," "id" PARAMS_N
(8)  PARAMS_N → ε
(9)  SEQUENCE → "expr" "EOL"
(10) SEQUENCE → "if" "expr" ":" "EOL" "INDENT" STATEMENTS "DEDENT" "else" ":" "EOL" "INDENT" STATEMENTS "DEDENT"
(11) SEQUENCE → "while" "expr" ":" "EOL" "INDENT" STATEMENTS "DEDENT"
(12) SEQUENCE → "id" INSTRUCT "EOL"
(13) SEQUENCE → "pass" "EOL"
(14) STATEMENTS → SEQUENCE SEQUENCE_N
(15) STATEMENTS → FUNC_RETURN SEQUENCE_N
(16) SEQUENCE_N → STATEMENTS
(17) SEQUENCE_N → ε
(18) SEQUENCE_N → "EOL" SEQUENCE_N
(19) INSTRUCT → "=" INSTRUCT_CONTINUE
(20) INSTRUCT → "(" TERM ")"
(21) INSTRUCT_CONTINUE → "expr"
(22) INSTRUCT_CONTINUE → "id" "(" TERM ")"
(23) TERM → "id" TERM_N
(24) TERM → TYPE TERM_N
(25) TERM → "None" TERM_N
(26) TERM → ε
(27) TERM_N → "," TERM_N_VALUE
(28) TERM_N → ε
(29) TERM_N_VALUE → "id" TERM_N
(30) TERM_N_VALUE → TYPE TERM_N
(31) TERM_N_VALUE → "None" TERM_N
(32) TYPE → "int"
(33) TYPE → "float"
(34) TYPE → "str"
(35) FUNC_RETURN → "return" RETURN_VALUE
(36) RETURN_VALUE → "EOL"
(37) RETURN_VALUE → "expr" "EOL"
```

Obrázek 2: LL gramatika

	"def"	"id"	"("	")"	"EOL"	"DEDENT"	"EOF"	","	"expr"	"if"	"while"	"pass"	"="	"None"	"int"	"float"	"str"	"return"
PROG	2	1			4		3		1	1	1	1						
PARAMS		5		6														
STATEMENTS		14							14	14	14	14						15
INSTRUCT			20										19					
PARAMS_N				8				7										
SEQUENCE		12							9	10	11	13						
SEQUENCE_N		16			18	17			16	16	16	16						16
FUNC_RETURN																		35
INSTRUCT_CONTINUE		22							21									
TERM		23		26										25	24	24	24	
RETURN_VALUE					36				37									
TERM_N				28				27										
TYPE															32	33	34	
TERM_N_VALUE		29												31	30	30	30	

Obrázek 3: LL tabulka

SIMPLIFIED PRECEDENT TABLE							
	+ -	* / //	(	)	RELATION	VALUE	$
+ -	>	<	<	>	>	<	>
* / //	>	>	<	>	>	<	>
(	<	<	<	=	<	<	
)	>	>		>	>		>
RELATION	<	<	<	>		<	>
VALUE	>	>		>	>		>
$	<	<	<		<	<	

Obrázek 4: Precedenční tabulka