Dokumentace k projektu do předmětů IFJ a IAL
Tým 031, varianta I

Adam Cologna	**(xcolog00)**	**27 %**
Tomáš Souček	(xsouce15)	33 %
Peter Pločica	(xploci01)	40 %
Jan Dejmal	(xdejma01)	0 %

Implementovaná rozšíření: **FUNEXP**

Obsah

1 Úvod

Cílem celého projektu je vytvořit program, který obdrží na standardní vstup kód v jazyce IFJ21 a na standardní výstup pošle mezikód v jazyce IFJcode21. Projekt je určen pro týmy se 3 nebo 4 členy.

2 Implementace

2.1 Lexikální analýza

V lexikálním analyzátoru probíhá prvotní zpracování vstupu. Funguje na principu konečného deterministického automatu, který prochází vstup znak po znaku a pomocí funkce `getToken` vrací jednotlivé lexémy.

Lexémy jsou reprezentovány strukturou `Token_t`, která obsahuje informace o typu lexému, jeho datech a délce jeho dat (sloužící pro správnou alokaci paměti). Funkce `getToken` se skládá ze tří podčástí. Nejdříve se zavolá funkce `getDraftToken`, která projde vstup a rozdělí jej na jednotlivé lexémy. Nad jejím výsledkem se zavolá funkce `idToKeyword`, která převede lexémy typu `id` na příslušný typ klíčového slova (např. pokud typ tokenu je „id"a jeho data obsahují „else“, převede typ tokenu na `keywordElse`).

Poslední částí je kontrola, jestli není výstupní token komentář. Pokud ano, zavolá se funkce `getToken` znovu, protože jedna z vlastností lexikálního analyzátoru je odfiltrovat části kódu, které nejsou podstatné pro další zpracování kódu.

Lexikální analýza se nachází v souboru `scanner.c`.

2.2 Syntaktická analýza

Syntaktický analyzátor má řídicí úlohu, jelikož po rozpoznání jazykové konstrukce volá příslušné sémantické procedury a následné generace kódu. Obsažena v souboru `parser.c`.

2.2.1 Metoda rekurzivního spádu

Metoda rekurzivního sestupu probíhá na základě pravidel LL gramatiky. Pro každé pravidlo existují různé kontroly, které se musí provést. Spoléháme na lexikální analýzu a získáváme tokeny pomocí `getToken()`. Kvalitní LL gramatika je klíčem k úspěšnému provedení projektu. Podle této části se řídí celý překlad a je nejdůležitější, jelikož byla zvolen překlad řízený syntaxí. .

2.2.2 Precedenční tabulka a analýza

Precedenční tabulka určuje jejími pravidly jaký derivační strom se má vytvořit. Její řádkové souřadnice reprezentují aktuální vrchol zásobníku a sloupcové souřadnice reprezentují vstupní symbol. Symbol dolaru značí dno zásobníku, na konci využívání precedenční tabulky bychom se měli dostat do stavu, že obdržíme dolar jak na vrcholu zásobníku i na vstupním symbolu (v tomto případě symbol dolaru indikuje konec vstupu).

V samotné tabulce symbol «"znační nutnost provést operaci, která vloží tento symbol a vstupní symbol na zásobník. Symbol »"znamená, že budeme redukovat obsah zásobníku podle předem stanovených pravidel.

2.3 Sémantická analýza

Pří sémantickém zpracování se využívá zásobník (nachází se v `expr.c`), do kterého se ukládají typy (I pro integer, F pro number, B pro bool, S pro string a N pro nil). Využívá se při deklaracích a typové kontrole. Po rozpoznání deklarace proměnné nebo parametru se volá funkce `void stInsertVar(symtable_t* table, char* name, char type, int blk_id)`. Tato funkce vloží do tabulky symbolů proměnnou `name` s typem `type`, který je získán z typového zásobníku a číslem bloku `blok_id`.

Parametry a lokální proměnné funkce mají číslo bloku 1 a pro každý další blok se číslo zvyšuje o 1. Spojením jména proměnné a čísla bloku vznikne jedinečné jméno využívané v generátoru kódu.

Po rozpoznání deklarace nebo definice funkce se volá funkce `void stInsertFunc(symtable_t* table, char* name, int blk_id, int defined, char* arg_type, char* ret_type, int built_in, int used)`, která do tabulky symbolů uloží funkci `name` s typy argumentů v řetězci `arg_type`, s typy návrtových hodnot v řetězci `ret_type`.

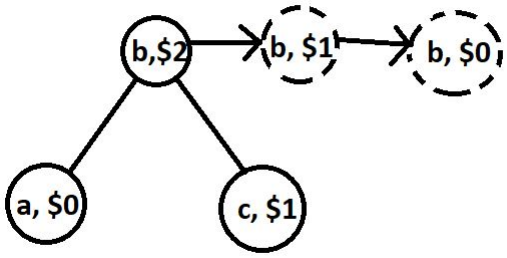
Pro vestavěné funkce je proměnná `built_in` nastavená na 1. Pro deklaraci funkce je proměnná `defined` nastavená na 0 a při její definici 1. Při použití proměnných a volání funkcí se využívá `bstNode_t* stFind(symtable_t* table, char* name)`, který umožňuje dávat chybová hlášení o chybějící definici nebo typové nekompatibilitě.

2.3.1 Tabulka symbolů

Tabulka symbolů je implementována pomocí binárního stromu. Vnitřní implementace vychází z 2. domácí úlohy z předmětu IAL, která byla doplněna o další nezbytnosti potřebné pro celkovou implementaci. Vyhledávání probíhá podle proměnné `name`, která reprezentuje argument identifikátoru.

Odlišností naší implementace je styl řešení vnořených proměnných. O vkládaných prvcích si uchováváme informaci o id bloku. Všechny prvky pak vkládáme do jednoho binárního stromu a pokud se stane, že musíme vložit proměnnou se stejným name, pak stávající přesuneme do hidden části nově vložené. Nově vkládaná proměnná pak v binární stromě nahradí stávající (tím vznikne propojení).

Časová náročnost vyhledávání v tabulce symbolů: O(logn) a nachází se jak bylo stanové dle zadání v `symtable.c`.



Obrázek 1: Zastínění, které bylo využito v tabulce symbolů

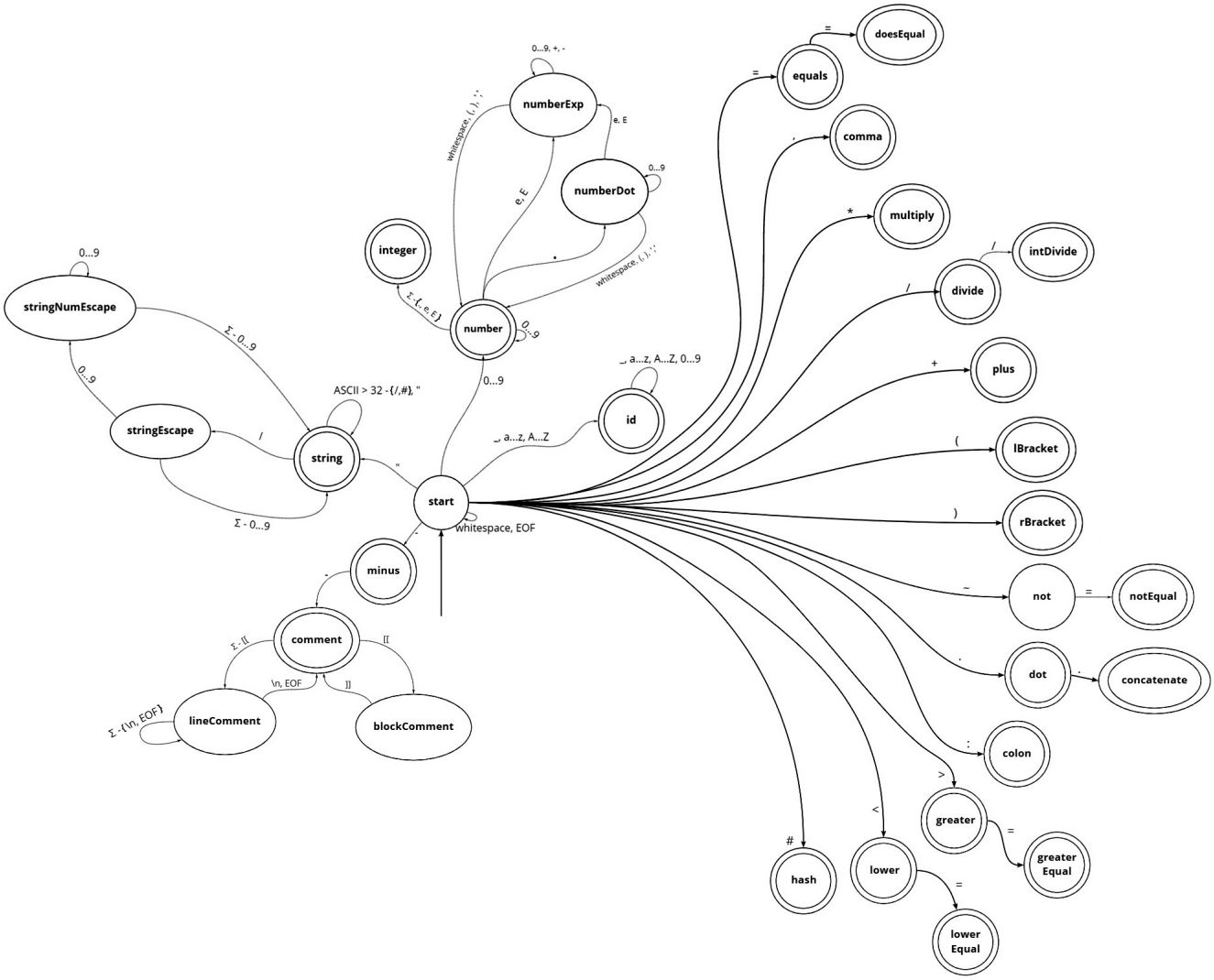
2.4 Generování kódu

Generování kódu je úzce spjato s sémantickou analýzou. Po úspěšném ověření určitých podmínek jsou zavolány funkce ze souboru `code.c`, které pomocí `printf` vypíšou na standardní výstup IFJcode21. Jediná výjimka je defvar, kde se musí projít celá funkce a provede se zabezpečení, že funkce skočí nejdřív na deklaraci funkce a až poté na tělo funkce.

3 Pravidla LL gramatiky

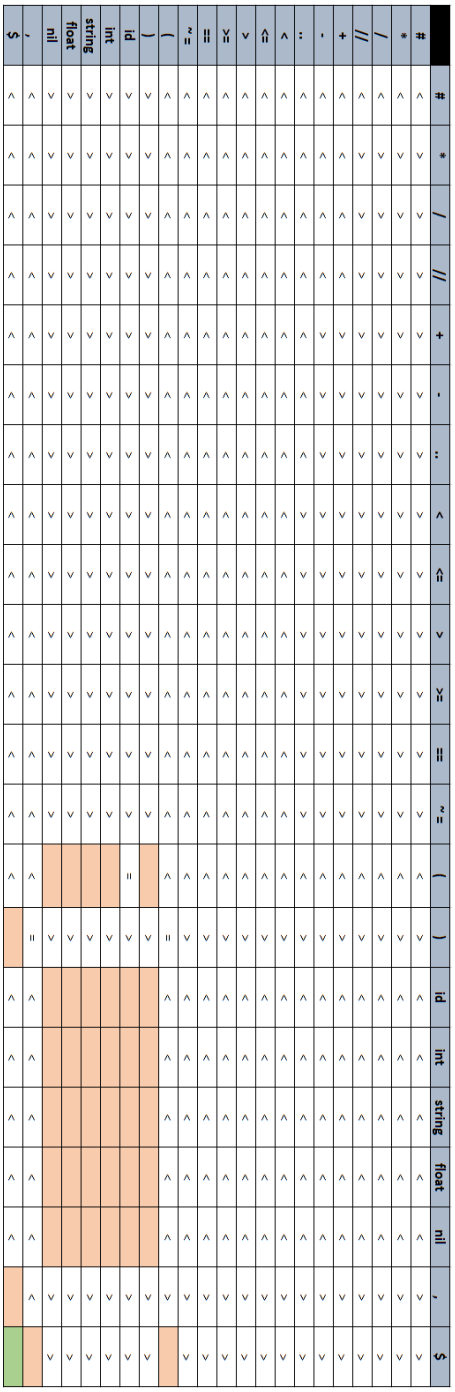
1. Program → Prolog (FunctionDeclaration | FunctionDefinition | FunctionCall)
2. Prolog → keywordRequire string
3. FunctionDeclaration → keywordGlobal id colon keywordFunction lBracket TypeList rBracket colon TypeList
4. FunctionDefinition → keywordFunction id lBracket ParameterList rBracket [colon TypeList] Block keywordEnd
5. FunctionCall → WriteCall | id lBracket ArgumentList rBracket
6. WriteCall → id lBracket Expression comma Expression rBracket
7. TypeList → [Type comma Type]
8. ParameterList → [id colon Type comma id colon Type]
9. ArgumentList → [Expression comma Expression]
10. Type → keywordInteger | keywordString | keywordNumber
11. Block → Statement
12. Statement → VariableDefinition | StatementIf | StatementWhile | AssignmentOrCall | StatementReturn
13. VariableDefinition → keywordLocal id colon Type [equals Expression]
14. StatementIf → keywordIf Expression keywordThen Block keywordElse Block keywordEnd
15. StatementWhile → keywordWhile Expression keywordDo Block keywordEnd
16. AssignmentOrCall → WriteCall | id (lBracket ArgumentList rBracket | comma id equals ExpressionList)
17. StatementReturn → keywordReturn ExpressionListOptional
18. ExpressionList → Expression comma Expression
19. ExpressionListOptional →[Expression comma Expression]
20. Expression → Expression (greaterEqual | lowerEqual | greater | lower | notEqual | doesEqual) Expression
21. Expression → Expression (plus | minus | multiply | intDivide | divide | concatenate) Expression
22. Expression → hash Expression
23. Expression → lBracket Expression rBracket
24. Expression → string | integer | number | keywordNil | id [lBracket ArgumentList rBracket]

4 Diagram konečného automatu



Obrázek 2: Diagram konečného automatu

5 Precedenční tabulka



	#	*	/	//	+	-	..	<	<=	>	>=	==	~=	(	)	id	int	string	float	nil	,	$
#	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
*	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
/	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
//	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
+	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
-	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
..	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
<=	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
>=	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
==	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
~=	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	>	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	<	<	
)	>	>	>	>	>	>	>	>	>	>	>	>	>	>		>					>	>
id	>	>	>	>	>	>	>	>	>	>	>	>	>	>		>					>	>
int	>	>	>	>	>	>	>	>	>	>	>	>	>	>		>					>	>
string	>	>	>	>	>	>	>	>	>	>	>	>	>	>		>					>	>
float	>	>	>	>	>	>	>	>	>	>	>	>	>	>		>					>	>
nil	>	>	>	>	>	>	>	>	>	>	>	>	>	>		>					>	>
,	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	<	<	
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<		<	<	<	<	<	

Obrázek 3: Precedenční tabulka

6 Závěr

6.1 Práce v týmu

Na začátku semestru jsme sestavili tým a rozdělili jsme si práci, ale bohužel jeden člen při vývoji se rozhodl nepokračovat v projektu a tudíž muselo dojít k přerozdělení práce. Byla to těžká situace a dostala nás do časového skluzu.

6.2 Rozdělení práce

- xcolog00 (Adam Cologna - vedoucí): Dokumentace, precedenční tabulka, testování.
- xsouce15 (Tomáš Souček): Lexikální analýza, tabulka symbolů, testování.
- xploci01 (Peter Pločica): Sémantická a syntaktická analýza, generování kódu.
- xdejma01 (Jan Dejmal): Hodnocen nula body, protože nám oznámil, že nebude na projektu pracovat.

6.3 Použitá prostředí a komunikační kanály

Komunikace v týmu probíhala vzhledem k aktuální epidemiologické situaci převážně pomocí online meetingů na Discordu, který sloužil i pro zanechávání zpráv jiným členům. Velmi naléhavé věci se řešily přes Messenger. Na začátku semestru pro rozdělení práce se využila možnost navštívit studentský klub.

Většina testování probíhala jak lokálně, tak i na serveru Merlin. Pro uchování a sdílení kódu byl zvolen Git a jako vývojové prostředí bylo převážně využito Visual Studio Code.