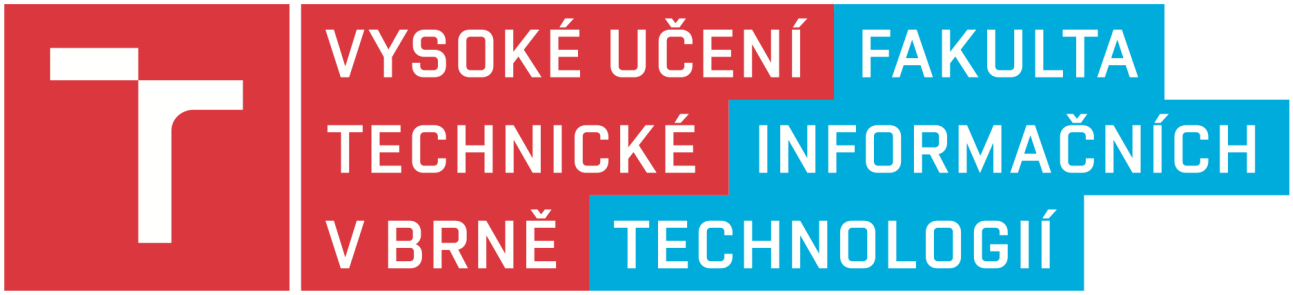


DOKUMENTACE PROJEKTU

IMPLEMENTACE PŘEKLADAČE IMPERATIVNÍHO JAZYKA IFJ18

Tým 091, varianta I

Jan Menšík (xmensi13) 30%
Jan Vaculík (xvacul30) 30%
Martin Trejtnar (xtrejt00) 30%
Patrik Leško (xlesko06) 10%

Listopad 2018

Obsah

Seznam tabulek

Seznam obrázků

1 Úvod

Tato dokumentace popisuje návrh a implementaci skupinového projektu do předmětů **Formální jazyky a překladače** (IFJ) a **Algoritmy** (IAL).

Projekt je implementován v jazyce C, načítá zdrojový kód zapsaný v jazyce IFJ18, jenž je zjednodušenou podmnožinou jazyka Ruby 2.0 a překládá jej do cílového jazyka IFJcode18 (mezikód). Zvolili jsme si **variantu I**, tedy implementaci tabulky symbolů pomocí binárního vyhledávacího stromu.

1.1 Vývojový cyklus a spolupráce v týmu

Práce na projektu započaly již na počátku měsíce října, bezprostředně po zveřejnění zadání a sestavení řešitelského týmu. Zprvu jsme se scházeli každou středu v prostorách fakulty. Cílem těchto sezení bylo zejména seznámení se se zadáním projektu.

Při řešení jsme využívali informací na stránkách projektu a postupovali dle načerpaných znalostí, které nám byly postupně předávány na přednáškách. Společně jsme vytvořili prvotní návrh stavového diagramu pro implementaci lexikální analýzy, později pravidla LL-gramatiky.

Postupem času, kdy jsme nabyli dostatek znalostí, probíhala sezení intenzivněji a stejně tak se otevíral i prostor pro samostatnou práci jednotlivých členů týmu.

Společná sezení však zůstala oblíbeným způsobem práce na projektu, jelikož jsme chtěli zužitkovávat znalosti všech členů týmu, maximálně urychlit řešení naskytnutých komplikací a zároveň se vyhnout problémum v komunikaci mezi jednotlivými moduly. Tento systém navíc efektivně umožňoval sledovat, jak práce pokračují. Taktéž jsme několikrát využili možnosti konzultací k projektu.

Procentuální bodové ohodnocení vychází z celkového množství odvedené práce a času věnovaného projektu.

Jan Menšík	**Primárně:**	Lexikální analýza, zpracování výrazů, organizace práce, ADT zásobník
	Podíl na:	Syntaktická a sémantická analýza, konzultace, generování instrukcí
Jan Vaculík	**Primárně:**	Syntaktická a sémantická analýza, generování instrukcí, Návrh a implementace tabulky symbolů
	Podíl na:	Lexikální analýza, konzultace, vestavěné funkce, LL gramatika
Martin Trejtnar	**Primárně:**	LL gramatika, zpracování výrazů, sepsání dokumentace, stavový diagram
	Podíl na:	Lexikální analýza, konzultace, LL gramatika, ADT zásobník
Patrik Leško	**Primárně:**	
	Podíl na:	Lexikální analýza, dokumentace, Návrh a implementace tabulky symbolů

1.2 Užitý software

Projekt jsme vyvíjeli v prostředí operačních systémů macOS, Ubuntu i Windows 10. Průběžně probíhali kontroly funkčnosti na školním severu CentOS Merlin. Při ladění jsme využívali možností programu OSS Valgrind. Stavový diagram byl vytvořen ve vektorovém grafickém editoru CorelDRAW X8 a samotná dokumentace vysázena v sázecím systému LaTeX.

2 Datové struktury

2.1 Tabulka symbolů

Tabulku symbolů jsme dle zadané varianty implementovali pomocí ADT binárního vyhledávacího stromu. Veškeré funkce pro práci s touto strukturou jsme implementovali rekurzivně. Při procházení stromu se rekurzivní sestup řídí porovnáváním klíčových řetězců daných záznamů a rozhoduje, zda sestoupí na levý, či pravý podstrom. Záznamy v tabulce symbolů jsou datové struktury nesoucí klíč, data a další pomocné proměnné. K implementaci tabulky symbolů jsme využili kostru projektu dostupnou na oficiálních stránkách projektu.

2.2 Zásobník

Implementace zásobníku byla potřeba pro implementování precedenční syntaktické analýzy (pro zpracování výrazů). Zásobník jsme však implementovali odlišným způsobem – jako jednosměrně vázaný lineární seznam. K této implementaci jsme se po vzájemné dohodě uchýlili z důvodu naší přehlednosti nad funkčností této struktury. Záznamy zásobníku na sebe odkazují ukazateli na následující záznam, přičemž samotný zásobník nese ukazatel na aktuální vrchol zásobníku. Pro práci se zásobníkem byly dále implementovány již typické funkce, např. pop, push, ad.

2.3 Další

Z kostry jsme využili ADT seznam instrukcí a datovou strukturu `string` pro práci s nekonečně dlouhými řetězci.

3 Lexikální analýza

Základním modulem lexikální analýzy je `scanner`.

3.1 Zvolený postup a implementace

Jedná se o implementaci deterministického konečného stavového automatu [1]. Jednotka pracuje jako jediná přímo se zdrojovým kódem. Scanner je reprezentován funkcí `int getNextToken()`, která přečte na požádání jednu lexému. Funkce je volána modulem `parser`. Úkolem scanneru je mimo jiné i zredukovat vstupní kód o dále nepodstatné části, jimiž jsou například komentáře. Funkce vrací datový typ `int`, který reprezentuje typ tokenu. Posloupnost znaků je funkcí `strAddChar` ukládána do pomocné proměnné `attr`.

Scanner je realizován konstrukcí `switch` v cyklu `while`, přičemž v každé iteraci čte znak ze standardního vstupu a dle daných pravidel přechází mezi jednotlivými stavy, dokud nepřečte jednu platnou lexému. V situacích, kdy dojde k přečtení znaku, který již patří dalšímu tokenu, se volá funkce `ungetc()`.

Pokud v nekoncovém stavu dojde k načtení neočekávaného znaku, jedná se o lexikální chybu. Je-li lexéma identifikována jako identifikátor, probíhá kontrola, zda se nejedná o klíčové slovo.

3.2 Problémy a překážky

V lexikální analýze bylo od začátku do konce prováděno významné množství úprav, jelikož jsme průběžně objevovali neošetřené situace. Například v prvnotním řešení modulu `scanner` jsme neuvažovali možnost, že by mezi operandy a operátory nemuseli být mezery. První výraznou změnou směru, jímž jsme se ubírali, bylo nalezení vzorové kostry projektu na stránkách projektu v archivu `jednoduchy_interpret`, podle níž jsme rozpracovaný projekt upravili a přepsali. Kostra nás nasměrovala v dalším postupu.

4 Syntaktická a sémantická analýza

Syntaktická a sémantická analýza je v režii modulu `parser`.

4.1 Rekurzivní syntaktická analýza shora dolů

Funkce `parse()` je volána přímo funkcí `main()`. Dále je funkcí `prog()` spuštěna metoda rekurzivního sestupu řízena pravidly LL gramatiky uvedenými níže [2]. Neterminály jsou rekurzivně volanými funkcemi, které kontrolují správný rozklad a dělí zdrojový kód dokud nedojde na zpracování tokenu typu `END_OF_FILE`. Průběžně probíhá generování instrukcí a kontrola syntaktických a sémantických chyb.

4.2 Problémy a překážky

Výrazné klopýtnutí a největší překážkou bylo obecné pochopení fungování syntaktické analýzy. Nejasnosti nám byly na konzultaci vyjasněny, sepsali jsme návrh LL gramatiky a práce mohly pokračovat.

Nejnáročnější vestavěnou funkcí byla dle našeho názoru funkce `Substr`, jelikož je realizována velkým počtem instrukcí z dostupné instrukční sady.

Nefunkčnost původního modulu `psa` vedla k přesunutí kódu precedenční syntaktické analýzy do těla modulu `parser`.

4.3 Precedenční tabulka operátorů

Pro zpracování výrazů jsme použili precedenční syntaktickou analýzu pracující zdola nahoru.

Precedence operátorů je řízena následující tabulkou, kde R zastupuje relační operátory (<, >, <=, >=, ==, !=), ID jsou hodnoty typů int, float, string a proměnné. Dvojice operátorů + a -, * a / jsou řízeny stejnými pravidly, proto jsou ve stejných sloupcích, respektive řádcích.

	+-	*/	ID	(	)	R	$
+-	>	<	<	<	>	>	>
*/	>	>	<	<	>	>	>
ID	>	>	×	×	>	>	>
(	<	<	<	<	=	<	×
)	>	>	×	×	>	>	>
R	<	<	<	<	>	>	>
$	<	<	<	<	×	<	✓

Legenda:

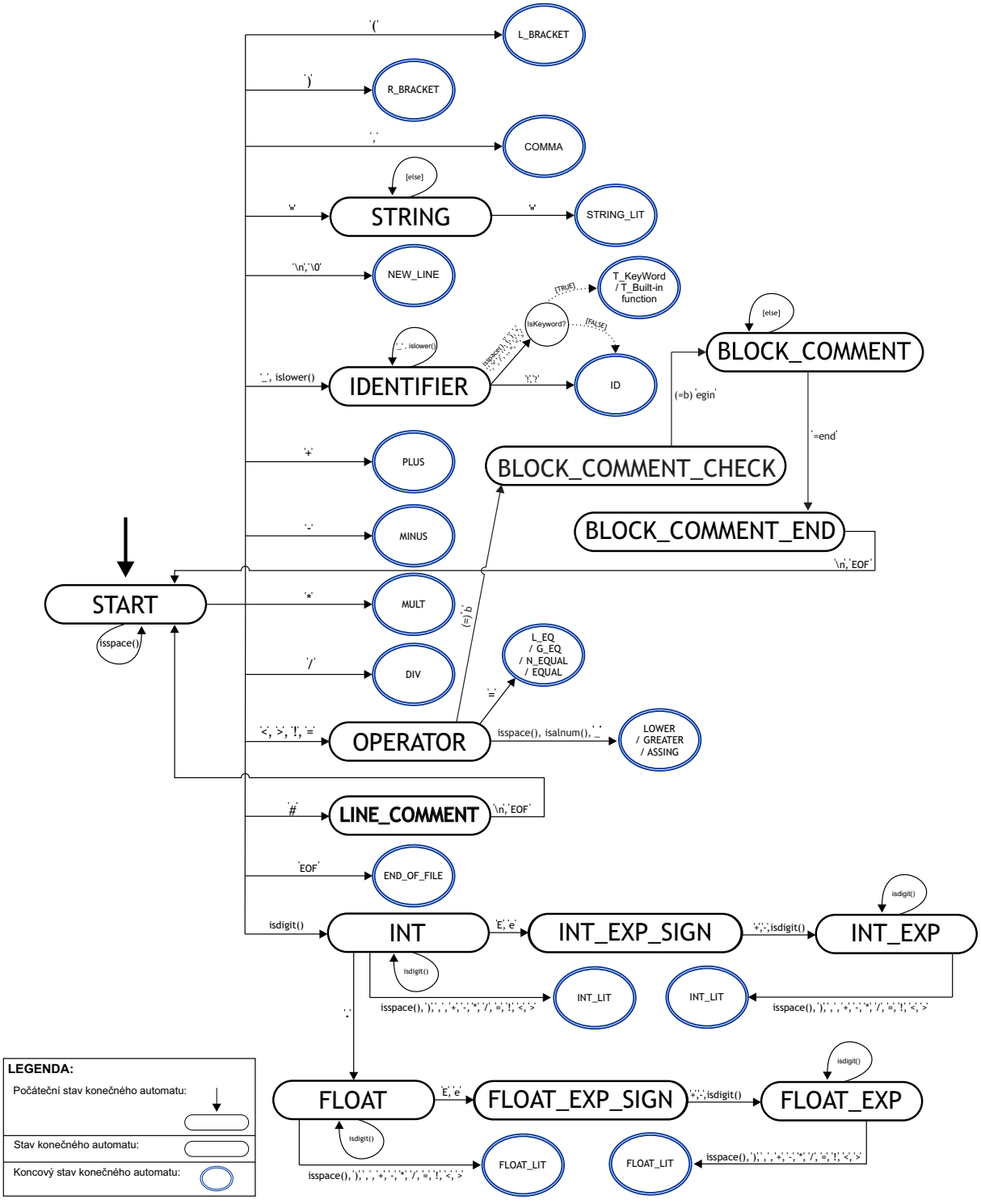
<	posun
>	redukce
=	rovnost (speciální případ redukce)
×	chyba
✓	přijetí výrazu

Tabulka 1: Precedenční tabulka operátorů

5 Generování instrukcí

Modul `interpret` slouží ke zpracování a výpisu instrukcí vygenerovaných v modulu `parser`. Volaný je v hlavním těle programu (main). Jednotlivé instrukce jsou uchovány ve struktuře `tListOfInstr`. Výsledkem tohoto modulu je tedy kompletní výpis tříadresných instrukcí na standardní výstup.

Neexituje-li v daném stavu přechodová funkce pro daný symbol, je tato situace klasifikována jako lexikální chyba. Tato skutečnost, z důvodu zachování přehlednosti, není v diagramu vyobrazena.



Obrázek 1: Stavový diagram

G = (N, T, P, S)
N = {<prog>, <statement>, <statList>,<expr>, <IDexpr>, <writeExpr>, <params>, <input>, <term>, <num>, <callFunction>, <otherParams>, <otherTerms>}
S = <prog >
T = {EOL, EOF, ID, WRITE, WHILE, DO, END, IF, THEN, ELSE, DEF, STR2INT, (, ,,), INT2CHAR, STRLEN, SUBSTR, INT_LIT, FLOAT_LIT, STRING_LIT, INPUTF, INPUTI, INPUTS }
P = {

1.	<prog>	→	<statList>	**EOF**						
2.	<statList>	→	<statement>	**EOL**	<statList>					
3.	<statList>	→	**ε**							
4.	<statement>	→	**ε**							
5.	<statement>	→	**ID**	<IDexpr>	<statement>					
6.	<statement>	→	**WRITE**	<writeExpr>						
7.	<statement>	→	**WHILE**	<expr>	**DO**	**EOL**	<statList>	**END**		
8.	<statement>	→	**IF**	<expr>	**THEN**	**EOL**	<statList>	**ELSE**	**EOL**	<statList>	**END**
9.	<statement>	→	**DEF**	**ID**	<params>	**EOL**	<statList>	**END**		
10.	<statement>	→	<input>							
11.	<statement>	→	**STR2INT**	**(**	<term>	**,**	<num>	**)**		
12.	<statement>	→	**INT2CHAR**	**(**	<num>	**)**				
13.	<statement>	→	**STRLEN**	**(**	<term>	**)**				
14.	<statement>	→	**SUBSTR**	**(**	<term>	**,**	<num>	**,**	<num>	**)**
15.	<statement>	→	**STR2INT**	<term>	**,**	<num>				
16.	<statement>	→	**INT2CHAR**	<num>						
17.	<statement>	→	**STRLEN**	<term>						
18.	<statement>	→	**SUBSTR**	<term>	**,**	<num>	**,**	<num>		
19.	<statement>	→	<callFunction>							
20.	<IDexpr>	→	<input>							
21.	<IDexpr>	→	**STR2INT**	**(**	<term>	**,**	<num>	**)**		
22.	<IDexpr>	→	**INT2CHAR**	**(**	<num>	**)**				
23.	<IDexpr>	→	**STRLEN**	**(**	<term>	**)**				
24.	<IDexpr>	→	**SUBSTR**	**(**	<term>	**,**	<num>	**,**	<num>	**)**
25.	<IDexpr>	→	**STR2INT**	<term>	**,**	<num>				
26.	<IDexpr>	→	**INT2CHAR**	<num>						
27.	<IDexpr>	→	**STRLEN**	<term>						
28.	<IDexpr>	→	**SUBSTR**	<term>	**,**	<num>	**,**	<num>		
29.	<IDexpr>	→	**INT_LIT**							
30.	<IDexpr>	→	**FLOAT_LIT**							
31.	<IDexpr>	→	**STRING_LIT**							
32.	<IDexpr>	→	<expr>							
33.	<IDexpr>	→	**ID**							
34.	<IDexpr>	→	<callFunction>							
35.	<writeExpr>	→	**(**	<term>	<otherTerms>	**)**				
36.	<params>	→	**ε**							
37.	<params>	→	**(**	<params>	**)**					
38.	<params>	→	**ID**	<otherParams>						
39.	<otherParams>	→	**,**	**ID**	<otherParams>					
40.	<otherParams>	→	**ε**							
41.	<input>	→	**INPUTF**							
42.	<input>	→	**INPUTI**							
43.	<input>	→	**INPUTS**							
44.	<input>	→	**INPUTF**	**(**	**)**					
45.	<input>	→	**INPUTI**	**(**	**)**					
46.	<input>	→	**INPUTS**	**(**	**)**					
47.	<term>	→	**ID**							
48.	<term>	→	**STRING_LIT**							
49.	<otherTerms>	→	**,**	<term>	<otherTerms>					
50.	<otherTerms>	→	**ε**							
51.	<num>	→	**ID**							
52.	<num>	→	**INT_LIT**							
53.	<callFunction>	→	**ID**	<params>						

}

Tabulka 2: Pravidla LL gramatiky

	ID	WRITE	WHILE	IF	DEF	STR2INT	INT2CHAR	STRLEN	SUBSTR	INT_LIT	FLOAT_LIT	STRING_LIT	(	,	INPUTF	INPUTI	INPUTS	ε
<prog>																		
<statList>																		3
<statement>	5	6	7	8	9	11, 15	12, 16	13, 17	14, 18									4
<IDexpr>	33					21, 25	22, 26	23, 27	24, 28	29	30	31						
<params>	38												37					36
<otherParams>														39				40
<input>															41, 44	42, 45	43, 46	
<term>	47											48						
<otherTerms>														49				50
<num>	51									52								
<callFunction>	53																	

Tabulka 3: LL tabulka

6 Použitá literatura

Reference

[1] Meduna, A., Lukáš, R.: *IFJ, přednášky [online].* Brno, Dostupné pro na URL: http://www.fit.vutbr.cz/study/courses/IFJ/public/materials/index.php

[2] Meduna, A.: *Elements of Compiler Design*. Taylor & Francis, New York, 2008, ISBN 9781420063233.