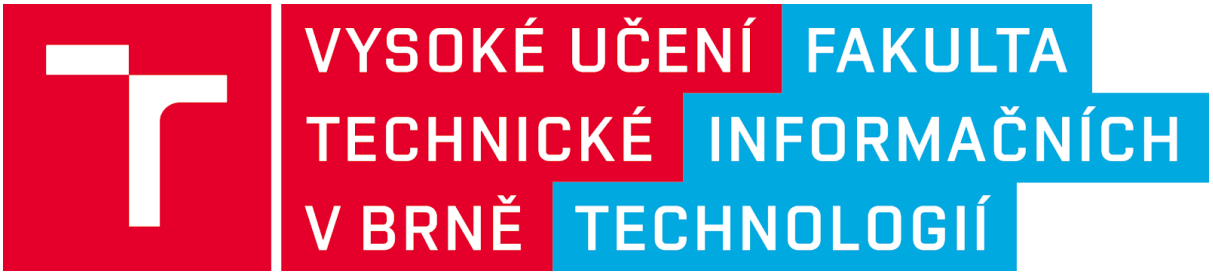
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Dokumentace k projektu do předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ20

Tým 023, varianta I

9. 12. 2020

Jiří Vlasák	xvlasa15	25%
Josef Kotoun	xkotou06	25%
Tomáš Beneš	xbenes55	25%
Vít Tlustoš	xtlust05	25%

1. Úvod **2**

2. Implementace **2**

2.1. Lexikální analýza 2

2.2. Syntaktická a sémantická analýza 2

2.3. Zpracování výrazů 2

2.4. Tabulka symbolů 3

2.5. Generátor kódu 3

2.6. Ostatní použité datové struktury 4

3. Práce v týmu **4**

3.1. Spolupráce 4

3.2. Rozdělení práce 4

4. Diagram konečného automatu lexikálního analyzátoru **5**

5. LL gramatika **6**

6. LL tabulka **8**

7. Precedenční tabulka **9**

1. Úvod

Dokumentace popisuje implementaci imperativního překladače jazyka IFJ20, který je podmnožinou jazyka GO. Výsledný program načítá ze standardního vstupu zdrojový kód jazyka IFJ20 a na standardní výstup vypisuje program v cílovém jazyce IFJcode20.

2. Implementace

2.1. Lexikální analýza

Lexikální analyzátor je implementován jako deterministický konečný automat. Hlavní funkce lexikální analýzy je get_token. Ta čte standardní vstup po znaku a pomocí přechodů do stavů automatu analyzuje, o jaký token se jedná. Návratová hodnota funkce je celé číslo které určuje, jestli byl nalezen validní token, nebo nastala lexikální chyba. V případě úspěchu je token uložen do struktury `token`, která obsahuje typ tokenu, pro určité typy tokenů jejich hodnotu a další informace, které jsou určeny pro výpis chyb, jako je např. číslo řádku, na kterém byl token přečten.

Pro rozeznání typu tokenu `keyword` od tokenu `identificator` je využita funkce `str_is_keyword`, která pro zadaný řetězec vrací logickou hodnotu podle toho, jestli se jedná o nějaké z platných klíčových slov.

Lexikální analyzátor je implementován v souboru `scanner.c` a v souboru `scanner.h` je uložena struktura tokenu a proměnné typu enum pro typ tokenu, klíčová slova a stavy automatu.

2.2. Syntaktická a sémantická analýza

Parser pracuje téměř s každým modulem projektu. Volá scanner kvůli získávání dalšího tokenu, parser výrazů kvůli zpracování výrazů, sémantickou tabulku pro sémantické kontroly, různé abstraktní datové typy pro jednodušší práci s daty (fronta, list) a generátor kódu.

Syntaktická analýza se řídí LL gramatikou a metodou rekurzivního sestupu podle LL tabulky, přičemž každé pravidlo z LL gramatiky je v kódu implementováno jako 1 funkce.

Přímo v těchto funkcích se provádí i sémantická analýza. Sémantická analýza kontroluje mnoho pravidel např. zda nedochází k redefinici funkce/proměnné, zda souhlasí typy a počet proměnných na levé straně s výrazy na pravé straně atd. Pro tyto pokročilé kontroly byla implementována abstraktní datová struktura fronta, která slouží k ukládání tokenů, typů a proměnných. Mezi jednotlivými funkcemi se posílá minimum parametrů, většina komunikace mezi funkcemi je řešena pomocí globálních proměnných.

2.3. Zpracování výrazů

Pro zpracování výrazů je použita precedenční analýza, která se řídí přiloženou precedenční tabulkou viz 7. Precedenční tabulka. Zpracování výrazu invokuje hlavní parser, který na základě LL gramatiky vyhodnotí, že se jedná o výraz a zavolá parser výrazů. Parser výrazů obdrží seznam tokenů, který představuje daný výraz.

Syntaktická kontrola - parser výrazu provádí syntaktickou kontrolu výrazu. V tabulce se to projeví tak, že není-li nalezeno pravidlo, kterým by se měl výraz redukovat, tak nastane syntaktická chyba.

Sémantická kontrola - parser výrazu kontroluje, zda se ve výrazu vyskytují shodné datové typy, dále je implementována kontrola na dělení nulou.

Generování kódu - v průběhu parsování je postupně voláno generování kódu.

2.4. Tabulka symbolů

Vzhledem k vybrané variantě zadání je tabulka symbolů implementována pomocí binárního stromu. Každý uzel binární stromu tabulky symbolů obsahuje jméno proměnné nebo funkce, typ uzlu (funkce nebo proměnná), ukazatele na levého a pravého potomka a ukazatel na strukturu s daty. Data uzlu jsou rozdílná pro funkci a pro proměnnou. Pro funkci je to např. počet parametrů, typy parametrů, návratové typy apod. Proměnná obsahuje pouze informaci o datovém typu.

Funkce jsou uloženy v globální tabulce symbolů. Každý blok ve funkci má svoji tabulku symbolů pro proměnné. Tabulky symbolů jsou zřetězeny pomocí dvousměrně vázaného seznamu. Každá položka seznamu obsahuje odkaz na tabulku symbolů, odkazy následující a předchozí položku seznamu a informaci o zanoření v blocích (scope index).

2.5. Generátor kódu

Generování kódu probíhá průběžně. Generátor má interní dynamický string, do kterého se postupně ukládají instrukce pro interpret. Tyto instrukce se vypíší až na konci práce překladu, aby se zbytečně nevypisoval kód, ve kterém je chyba.

Hned na začátku generování se do stringu přidají instrukce potřebné pro běh každého programu. Například vestavěné funkce, globální proměnné atd...

V main funkci se vytvoří první rámec a při volání jakékoliv funkce se se aktuální rámec uloží do zásobníku a vytvoří se nový. Po ukončení funkce se opět rámec ze zásobníku vytáhne.

Při práci s proměnnými se konkatenuje název proměnné a unikátní číslo scopu ve kterém se proměnná nachází, aby se zajistila práce se správnou proměnnou.

Všechny proměnné jsou definovány na konci funkce, a to proto aby se zajistilo že se bude každá proměnná definovat pouze jednou. V těle funkce se s těmito proměnnými pouze pracuje.

Do funkcí se předávají parametry přes zásobník. Před voláním funkce se všechny parametry uloží do zásobníku a v těle volané funkce se proměnné vytáhnou v opačném pořadí. Stejně se předávají i návratové hodnoty funkce.

Veškeré zpracování výrazů a porovnávání se děje na zásobníku. Parser zařizuje, aby se do zásobníku vložili potřebné proměnné a hodnoty ve správném pořadí. Také zařizuje užití operací ve správném pořadí.

Při generování ifů a forů, volá parser funkce generátoru. Generátor generuje návěští s skoky na návěští v takovém pořadí, aby se zajistila správná funkčnost.

2.6. Ostatní použité datové struktury

Při implementaci překladače jsme využili několik datových struktur. Jednou z nich je dynamický řetězec, jehož implementace se nachází v souboru `str.c`. Využili jsme již hotovou implementaci z interpretu jednoduchého jazyka umístěného na veřejných stránkách IFJ.
Dále jsme pro tabulku symbolů využili binární strom a dvousměrně vázaný seznam. Implementace těchto struktur jsme z části převzali z domácích úkolů z předmětu Algoritmy a upravili je pro naše potřeby. Jejich implementace je v zdrojových souborech `symtable.c` a `dl_list.c`.
Pro potřebu ukládání tokenů v rámci sémantické analýzy byla implementována fronta tokenů, jejíž implementace je ve zdrojovém souboru `queue.c`. Implementace fronty byla také převzata z domácích úkolů z předmětu Algoritmy.
Při precedenční analýze bylo třeba využít zásobník, jehož implementaci lze nalézt v souboru `expression_stack.c`

3. Práce v týmu

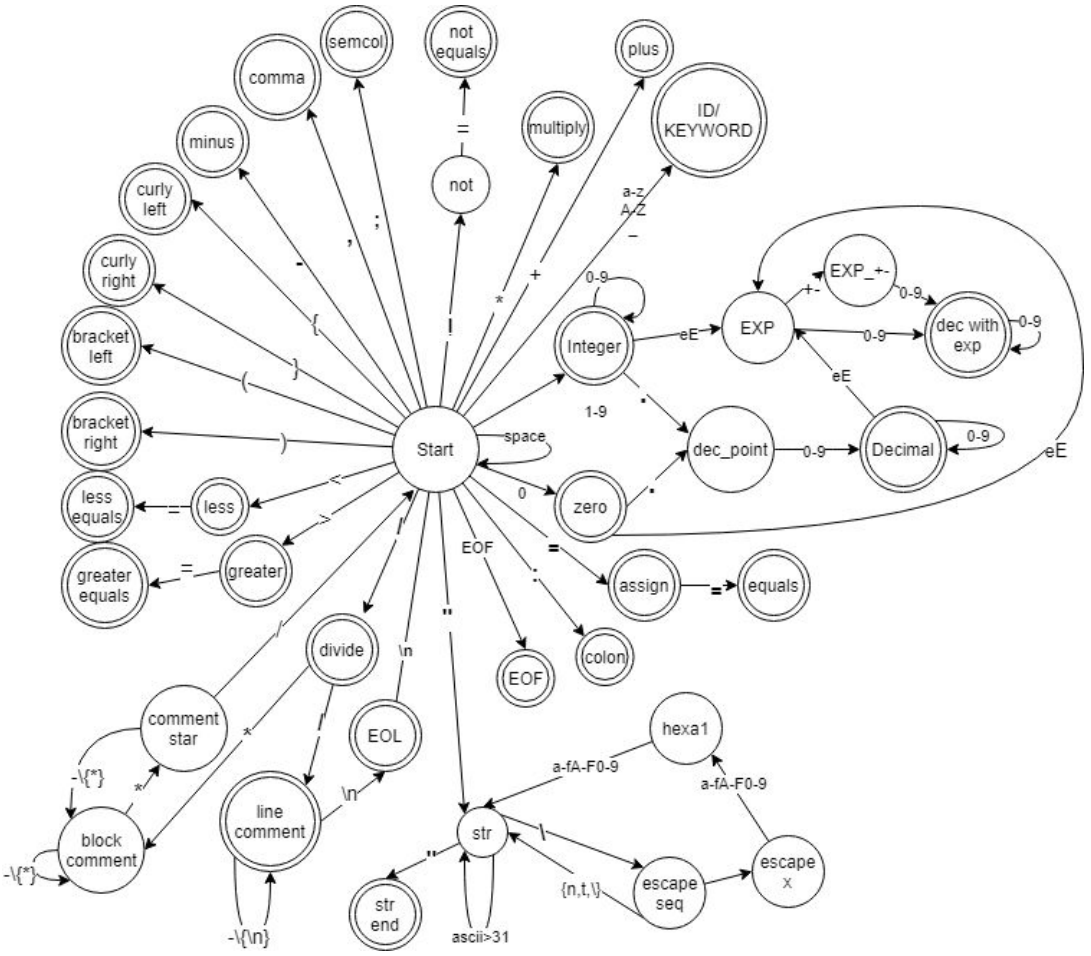
3.1. Spolupráce

Práce v týmu byla bezproblémová. Ve stejném složení jsme již spolupracovali na projektu v předmětu IVS. Pro komunikaci jsme využívali messenger a discord. Pravidelně jsme se scházeli na discordu a diskutovali problémy v implementaci a propojení jednotlivých komponent programu. Jako verzovací systém jsme využili git hostovaný na privátním repozitáři na GitHubu.

3.2. Rozdělení práce

Jiří Vlasák (xvlasa15): Syntaktická a sémantická analýza, návrh LL gramatiky, testování, dokumentace
Josef Kotoun (xkotou06): Lexikální analýza, tabulka symbolů, testování, dokumentace
Vít Tlustoš (xtlust05): Zpracování výrazů precedenční analýzou, návrh LL gramatiky, testování, dokumentace
Tomáš Beneš (xbenes55): Generátor kódu, dokumentace

4. Diagram konečného automatu lexikálního analyzátoru



5. LL gramatika

1. <prog> -> <prolog> EOL <eols> <func_decl> <func_list> EOF
2. <eols> -> EOL <eols>
3. <eols> -> ε
4. <prolog> -> PACKAGE ID
5. <func_list> -> <func_decl> <eols> <func_list>
6. <func_list> -> ε
7. <func_decl> -> FUNC ID (<param_first>) <return_list> <body>
   - 7.1. <param_first> -> ε
   - 7.2. <param_first> -> <param> <param_n>
   - 7.3. <param_n> -> , <param> <param_n>
   - 7.4. <param_n> -> ε
   - 7.5. <param> -> ID <type>
   - 7.6. <return_list> -> (<type_first>)
   - 7.7. <return_list> -> ε
   - 7.8. <type_first> -> <type> <type_n>
   - 7.9. <type_first> -> ε
   - 7.10. <type_n> -> , <type> <type_n>
   - 7.11. <type_n> -> ε
8. <type> -> STRING
9. <type> -> INT
10. <type> -> FLOAT64
11. <body> -> { <eols> <statement_list> <eols> }
12. <statement_list> -> <statement> EOL <eols> <statement_list>
13. <statement_list> -> ε
14. <statement> -> <literal_expr>
15. <statement> -> ID <id_list> <statement_action>
    - 15.1. <id_list> -> , ID <id_list>
    - 15.2. <id_list> -> ε
    - 15.3. <statement_action> -> = <statement_value>
      - 15.3.1. <statement_value> -> ID <arg_expr>
        - 15.3.1.1. <arg_expr> -> (<first_arg>)
        - 15.3.1.2. <arg_expr> -> <expr_end> <expr_n>
      - 15.3.2. <statement_value> -> <literal_expr> <expr_n>
        - 15.3.2.1. <expr_n> -> , <expr>
        - 15.3.2.2. <expr_n> -> ε
    - 15.4. <statement_action> -> (<first_arg>)
      - 15.4.1. <first_arg> -> ε
      - 15.4.2. <first_arg> -> ID <arg_n>
      - 15.4.3. <arg_n> -> , ID <arg_n>
      - 15.4.4. <arg_n> -> ε
    - 15.5. <statement_action> -> := <expr>

16. <statement> -> IF <expr> <body> ELSE <eols> <body>
17. <statement> -> FOR <definition> ; <expr> ; <assignment> <body>
    - 17.1. <definition> -> ID := <expr>

17.2. <definition> -> ε
17.3. <assignment> -> ID = <expr>
17.4. <assignment> -> ε
18. <statement> -> RETURN <statement_value>

19. <literal_expr> -> <literal> <expr_end>
20. <literal_expr> -> (<expr>) <expr_end>

21. <expr> -> (<expr>) <expr_end>
22. <expr> -> <term> <expr_end>
22.1. <term> -> ID
22.2. <term> -> <literal>
22.2.1. <literal> -> INT_LITERAL
22.2.2. <literal> -> STRING_LITERAL
22.2.3. <literal> -> FLOAT_LITERAL
22.3. <expr_end> -> ε
22.4. <expr_end> -> <binary_operator> <expr>
22.4.1. <binary_operator> -> +
22.4.2. <binary_operator> -> -
22.4.3. <binary_operator> -> *
22.4.4. <binary_operator> -> /
22.4.5. <binary_operator> -> <
22.4.6. <binary_operator> -> >
22.4.7. <binary_operator> -> <=
22.4.8. <binary_operator> -> >=
22.4.9. <binary_operator> -> ==
22.4.10. <binary_operator> -> !=

6. LL tabulka

	EOF	EOL	ID	(	)	INT	,	STRING	FLOAT64	{	}	=	:=	PACKAGE	FUNC	IF	ELSE	FOR	;	RETURN
<prog>														1						
<eols>	3	2	3	3						3	3				3	3		3		3
<prolog>														4						
<func_list>	6														5					
<func_decl>															7					
<param_first>			7.2		7.1															
<param_n>					7.4		7.3													
<param>			7.5																	
<return_list>				7.6						7.7										
<type_first>					7.9	7.8		7.8	7.8											
<type_n>					7.11		7.10													
<type>						9		8	10											
<body>										11										
<statement_list>		13	12	12							13					12		12		12
<statement>			15	14												16		17		18
<id_list>					15.2		15.1					15.2	15.2							
<statement_action>				15.4								15.3	15.5							
<statement_value>			15.3.1	15.3.2																
<arg_expr>				15.3.1.1																
<expr_n>		15.3.2.2					15.3.2.1													
<first_arg>			15.4.2		15.4.1															
<arg_n>					15.4.4		15.4.3													
<definition>			17.1																17.2	
<assignment>			17.3							17.4										
<literal_expr>				20																
<expr>			22	21																
<term>			22.1																	
<literal>																				
<expr_end>		22.3			22.3		22.3			22.3									22.3	
<binary_operator>																				

	INT_LITERAL	STRING_LITERAL	FLOAT_LITERAL	+	-	*	/	<	>	<=	>=	==	!=
<prog>													
<eols>	3	3	3										
<prolog>													
<func_list>													
<func_decl>													
<param_first>													
<param_n>													
<param>													
<return_list>													
<type_first>													
<type_n>													
<type>													
<body>													
<statement_list>	12	12	12										
<statement>	14	14	14										
<id_list>													
<statement_action>													
<statement_value>	15.3.2	15.3.2	15.3.2										
<arg_expr>				15.3.1.2	15.3.1.2	15.3.1.2	15.3.1.2	15.3.1.2	15.3.1.2	15.3.1.2	15.3.1.2	15.3.1.2	15.3.1.2
<expr_n>													
<first_arg>													
<arg_n>													
<definition>													
<assignment>													
<literal_expr>	19	19	19										
<expr>	22	22	22										
<term>	22.2	22.2	22.2										
<literal>	22.2.1	22.2.2	22.2.3										
<expr_end>				22.4	22.4	22.4	22.4	22.4	22.4	22.4	22.4	22.4	22.4
<binary_operator>				22.4.1	22.4.2	22.4.3	22.4.4	22.4.5	22.4.6	22.4.7	22.4.8	22.4.9	22.4.10

7. Precedenční tabulka

Vrchol/Vstup	{ +, - }	{ *, / }	(	)	{ ==, !=, < , <=, >, >= }	{ id, int, string, float64 }	$
{ +, - }	>	<	<	>	>	<	>
{ *, / }	>	>	<	>	>	<	>
(	<	<	<	=	<	<	∅
)	>	>	∅	>	>	∅	>
{ ==, !=, <, <=, >, >= }	<	<	<	>	∅	<	>
{ id, int, string, float64 }	>	>	∅	>	>	∅	>
$	<	<	<	∅	<	<	∅