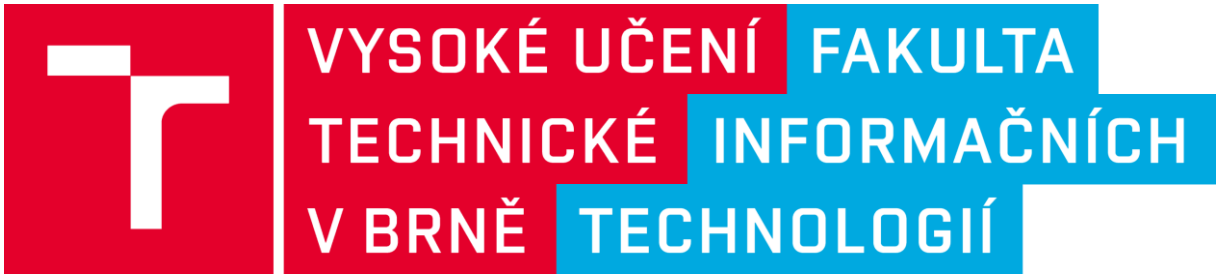


Dokumentace návrhu a implementace

Projekt implementace překladače imperativního jazyka IFJ22
Tým xpavli95

Nebyla implementována žádná rozšíření

Pavlíček Jan (xpavli95) – Vedoucí týmu	25%
Nekula Štěpán (xnekul04)	25%
Vadovič Matej (xvadov01)	25%
Soukup Martin (xsouku15)	25%

Obsah

1. Motivace projektu

Vytvořit program v jazyce C, který načte zdrojový kód jazyka IFJ22 a přeloží jej do cílového jazyka IFJcode22. Samotný jazyk je zjednodušenou podmnožinou jazyka PHP.

2. Návrh a implementace

Program byl rozdělen na pět spolu komunikujících částí.

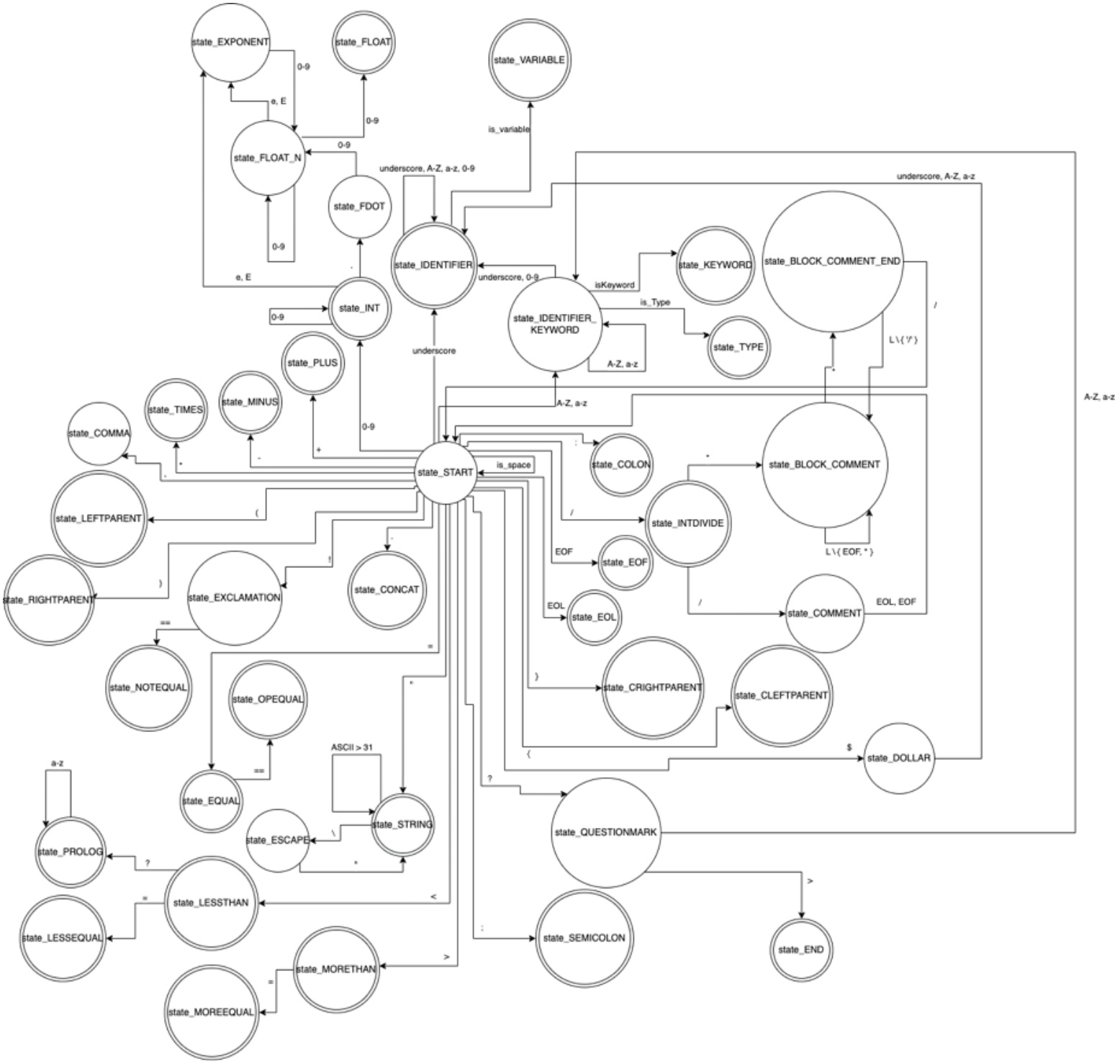
- Lexer (scanner.c/h)
- Rekurzivní sestup (recursive_descent.c/h)
- Precedenční analýza (precedent.c/h)
- Sémantická analýza (semantic.c/h)
- Generování kódu (code_generator.c/h)

Hlavním tělo programu se nachází v souboru main.c a volá rekurzivní sestup, který poté řídí celý průběh překladu.

2.1. Lexikální analýza

Lexikální analýza (dále Scanner) je implementována pomocí konečného stavový automatu. Hlavní funkce getToken přebírá jako parametry strukturu Token a soubor se zdrojovým kódem načítaný ze standardního vstupu.

Scanner postupně prochází znaky a podle vlastností jazyka IFJ22 tvoří lexémy. Jakmile se scanner dostane do konečného stavu a lexém má správný tvar, přiřadí se mu typ konečného stavu. Pokud scanner odhalí lexikální chybu (neplatný znak, špatný tvar lexému), skončí s návratovou hodnotou 1. Speciální případy nastanou, pokud je špatně zapsán prolog zdrojového kódu nebo uzavírací značka, kdy scanner končí s návratovou hodnotou 2.



Obrázek 1: Diagram konečného automatu

2.2. Rekurzivní sestup

Rekurzivní sestup je základní část syntaktické analýzy, provádí syntaktickou kontrolu jednotlivých částí zdrojového kódu, provádí postupné vytváření AST, který poté předává do sémantické analýzy na další modifikování. Ke svému průchodu volá postupně scanner a jednotlivé tokeny ze scanneru ukládá do zásobníku, s tímto zásobníkem tokenů poté pracuje po celou dobu rekurzivního sestupu. V případě, že narazí na výraz, volá precedenční analýzu. Pravidla pro LL-gramatiku jsou tvořena ručně, ke generování LL-tabulky byl využit online nástroj.

```
1  <prolog> -> <?php declare(strict_types=1); <prog>
2  <prog> -> function id ( <param> ) <ret_type> { <body> } <prog>
3  <prog> -> id ( <arg> ) ; <prog>
4  <prog> -> $id = <assign> ; <prog>
5  <prog> -> if ( <expr> ) { <body> } else { <body> } <prog>
6  <prog> -> while ( <expr> ) { <body> } <prog>
7  <prog> -> return <return_n> ; <prog>
8  <prog> -> <expr> ; <prog>
9  <prog> -> ?>
10 <prog> -> EOF
11 <param> -> <type> $id <next_param>
12 <param> -> ε
13 <next_param> -> , <type> $id <next_param>
14 <next_param> -> ε
15 <arg> -> <term> <next_arg>
16 <arg> -> ε
17 <next_arg> -> , <term> <next_arg>
18 <next_arg> -> ε
19 <term> -> $id
20 <term> -> term_int
21 <term> -> term_float
22 <term> -> null
23 <term> -> term_string
24 <ret_type> -> : <type>
25 <ret_type> -> ε
26 <body> -> id ( <arg> ) ; <body>
27 <body> -> if ( <expr> ) { <body> } else { <body> } <body>
28 <body> -> while ( <expr> ) { <body> } <body>
29 <body> -> $id = <assign> ; <body>
30 <body> -> return <return_n> ; <body>
31 <body> -> <expr> ; <body>
32 <body> -> ε
33 <assign> -> <expr>
34 <assign> -> id ( <arg> )
35 <return_n> -> <expr>
36 <type> -> ?string
37 <type> -> string
38 <type> -> ?int
39 <type> -> int
40 <type> -> ?float
41 <type> -> float
```

Obrázek 2: LL-gramatika

	<?php declare(strict_types=1);	function	id	(	)	{	}	$id	=	if	expr	else	while	return	?>	EOF	ε	,	term_int	term_float	null	term_string	:	?string	string	?int	int	?float	float	$
prolog	1																													
prog		2	3					4		5	8		6	7	9	10														
param																	12							11	11	11	11	11	11	
next_param																	14	13												
arg								15									16		15	15	15	15								
next_arg																	18	17												
term								19											20	21	22	23								
ret_type																	25						24							
body			26					29		27	30		28				32													
assign			34								33																			
return_n											35																			
type																								36	37	38	39	40	41	

Obrázek 3: LL-tabulka

2.3. Precedenční analýza

Precedenční analýza je druhá část syntaktické analýzy. Volá se v případě, kdy rekurzivní sestup dojde k výrazu. Precedenční analýza poté postupně kontroluje syntaxi zadaného výrazu. Vytváří postfixovou notaci výrazu, pokud není nalezena žádná chyba ve výrazu, precedenční analýza vytvoří kus abstraktního syntaktického stromu odpovídající danému výrazu.

$E \rightarrow E * E$ $E \rightarrow E.E$ $E \rightarrow E <= E$ $E \rightarrow E! == E$
$E \rightarrow E/E$ $E \rightarrow E < E$ $E \rightarrow E >= E$ $E \rightarrow (E)$
$E \rightarrow E + E$ $E \rightarrow E > E$ $E \rightarrow E === E$ $E \rightarrow id$
$E \rightarrow E - E$

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

Obrázek 4: Precedenční tabulka

2.4. Sémantická analýza

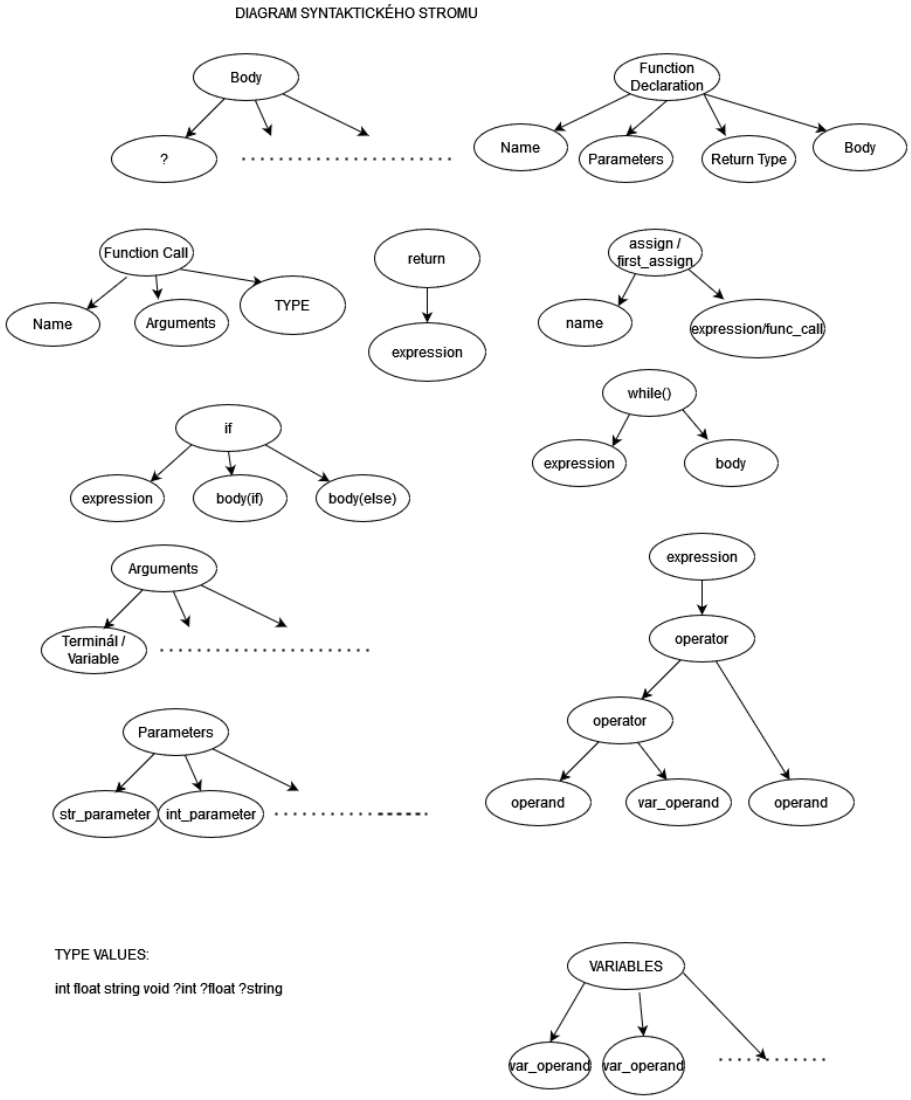
Sémantická analýza je rozdělena do dvou částí.

První část kontroluje nedefinované funkce a pokus o redefinici funkcí, kontrolu počtu parametrů při volání funkce a počet výrazů při návratu z funkce.

Řešení je použití tabulky symbolů. Analýza používá dva zásobníky tabulek symbolů. První ze zásobníků slouží pro uložení záznamů o definicích funkcí (návratový typ, počet a typ parametrů). Naplnění tabulky funkcí se vykoná v prvním průchodu abstraktního syntaktického stromu (dále AST).

Druhá tabulka symbolů slouží pro záznam proměnných v jednotlivých rámcích zdrojového kódu. Při druhém průchodu AST sémantická analýza vytvoří rámce tabulky symbolů, do kterých se ukládají záznamy o použitých proměnných. Informace o proměnných jsou poté uloženy do AST. První rámec zaznamenává hlavní tělo programu, další rámce jsou lokální pro dané funkce. Lokální rámec existuje jen dokud je zpracovávané tělo funkce, při návratu do hlavního těla je uvolněn a aktivním rámce je rámec hlavního těla.

Druhá část zabezpečuje kontrolu nedefinovaných proměnných, správných typů argumentů funkcí a typů návratových hodnot. Protože tato část nemůže kontrolovat věci za běhu programu, je tato povinnost přenesena na modul generování kódu.



Obrázek 5: Diagram AST

2.5. Generování kódu

Generátor kódu je samostatný modul, jenž vytvoří zdrojový kód na základě AST, který je předán syntaktickou analýzou. Při generování kódu je již předpokládáno, že AST je validní až na chyby kontrolované za běhu programu.

- Proměnné – na začátku hlavního těla jsou definovány, kvůli dynamické kontrole inicializace před použitím ve výrazu. Uživatelské proměnné jsou definovány jako proměnné momentálního lokálního rámce. Pro potřeby překladu bylo vytvořeno několik pomocných globálních proměnných.
- Výrazy – implementace výrazů pomocí postupného postorder procházení AST. Operandy jsou přidány do zásobníku a operátory použity v zásobníkovém tvaru.
- Funkce – na začátku funkce je vytvořen vlastní lokální rámec pro lokální proměnné. Vestavěné funkce jsou některé předdefinovány na začátku programu, u funkcí, kde byla možnost, jsou implementovány stylem volání inline funkce jazyka C. Argumenty funkcí jsou předávány přes zásobník.
- Cykly a podmínky – pro potřeby zajištění unikátnosti názvu návěští má každý výskyt podmínky nebo cyklu přiřazen unikátní číslo, které je součástí jejich návěští.

3. Použité algoritmy a struktury

3.1. Dynamický řetězec

Struktura řetězců (strings.c/h) obsahuje pole znaků, délku pole a velikost pole v paměti. Struktura je použita pro práci s lexémy, kdy v modulu Scanner skládá znaky za sebe a tím tvoří konečnou hodnotu Tokenu.

3.2. Zásobníky

Překladač používá pro překlad zásobníky (token_stack.c/h) tokenů a zásobník tabulek symbolů (symtable.c/h) .

Zásobníky tokenů se používají pro průchod rekurzivním sestupem, kdy se tokeny skládají za sebe pro jednoduší kontrolu syntaxe, do stejného zásobníku se poté ukládají i výrazy v postfixové notaci.

Zásobníky tabulek symbolů jsou použity v rámci sémantické analýzy.

3.3. Tabulka symbolů

Tabulka symbolů (symtable.c/h) slouží pro ukládání informací o funkcích a proměnných v sémantické kontrole. Je implementována jako tabulka rozptýlených prvků. Pro práci s tabulkami symbolů je tvořen zásobník ukazatelů na jednotlivé tabulky. Tabulka sestává z mapovacího pole a zřetězených listů záznamu. Každý záznam obsahuje odkaz na data a odkaz na další záznam ve zřetězeném listu. Struktura dat sestává z klíče a odkazu na strukturu, obsahující informace spojené s proměnnou nebo funkcí. O funkcích se ukládá počet a typ parametrů a typ návratové hodnoty. O proměnných jejich datový typ, což není využito z důvodu kontroly typu za běhu programu. Tabulka symbolů nám slouží pro kontrolu nedefinovaných/redefinovaných funkcí, nedefinovaných proměnných a kontrolu počtu argumentů při volání funkce.

3.4. AST

Abstraktní syntaktický strom (syntax_tree.c/h) se vytváří v průběhu syntaktické analýzy, hlavní část se tvoří v rámci rekurzivního sestupu, výrazy poté v rámci precedenční analýzy v postfixové notaci.

4. Rozdělení práce

Podle jednotlivých částí byla rozdělena práce na:

- Lexikální analýza – xsouku15
- Syntaktická analýza – xpavli95
- Sémantická analýza – xvadov01
- Generátor kódu – xnekul04

4.1. Podrobnější rozdělení

- xsouku15 – lexikální analýza, Strings, dokumentace
- xpavli95 – syntaktická analýza, vedení týmu, správa repozitáře
- xvadov01 – sémantická analýza, symtable
- xnekul04 – generování kódu, symtable, struktura AST

5. Vývojový cyklus

Vývoj byl započat hned po zveřejnění zadání. Začalo se s konečným automatem, který se převedl na skener, neboť jeho implementace nezávisí na dalších modulech, a zároveň implementací teoretické části: LL tabulka a precedenční tabulka. Také byla vytvořena tabulka symbolů pro generické položky pro pozdější použití.

Po dokončení teoretické části se začalo se syntaktickou analýzou, která navazovala na skener a vycházela z LL tabulky a precedenční tabulky.

Po domluvě na komunikaci mezi syntaktickou analýzou, sémantickou analýzou a generátorem kódu ve stylu AST mohl vývoj zmíněných modulů také začít.

Před vznikem prvního prototypu byl každý modul otestován samostatně v simulovaném prostředí, aby byla zaručena alespoň částečná funkčnost.

Po složení všech modulů dohromady a dokončení prvního funkčního prototypu jsme pokračovali vývoj iterativní metodou na základě zpětné vazby ze studentských testů.

6. Způsob práce v týmu

Přestože jeden z členů týmu je na zahraničním pobytu a osobní schůzka celého týmu nebyla možná, jsme se pravidelně scházeli online skoro každý víkend po celou dobu vývoje. Na schůzce byl vždy probrán dosavadní pokrok a co bude dále potřeba udělat. V případě potřeby proběhla týmová konzultace problému, který bylo třeba vyřešit.

V případě potřeby byl projekt diskutován i v textové podobě během týdne.

Jako hlavní nástroje byl používán verzovací systém Git s hostingem na GitHubu a komunikace především probíhala na platformě Discord.

7. Zdroje

Pro generování LL-tabulky byl použit nástroj
https://jsmachines.sourceforge.net/machines/ll1.html