VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace projektu z předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ22

Tým xbarto0g, varianta – TRP

Petr Bartoš – xbarto0g (vedoucí)	40 %
Tomáš Rajsigl – xrajsi01	30 %
Lukáš Zedek – xzedek03	30 %
Dmytro Afanasiev – xafana01	0 %

2. prosince 2022

1 Úvod

Cílem projektu je vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ22 a přeloží jej do cílového jazyka IFJcode22. Jazyk IFJ22 je zjednodušenou podmnožinou jazyka PHP. Konkrétně se jedná o variantu zadání s implementací tabulky symbolů pomocí tabulky s rozptýlenými položkami.

2 Práce v týmu

Práci jsme si rozdělili jakožto čtyřčlenný tým, avšak z důvodu nespolupracování a nezájmu jednoho člena jsme byli nuceni si práci přerozdělit a projekt vypracovat pouze ve třech. Vzhledem k časové náročnosti a složitosti daných částí jsme se rozhodli pro nerovnoměrné rozdělení bodů.

2.1 Rozdělení práce

- Petr Bartoš
  - Návrh gramatik pro syntaktickou analýzu, syntaktická analýza, sémantická analýza, tabulka symbolů, chybové hlášky, kostra pro testování
- Tomáš Rajsigl
  - Návrh konečného automatu a gramatik pro syntaktickou analýzu, lexikální analýza, testování parseru, dokumentace a prezentace
- Lukáš Zedek
  - Generování cílového kódu, testování jednotlivých modulů překladače, Makefile, dokumentace a prezentace

Kontrole kódu, jeho čitelnosti a opravě chyb jsme se věnovali všichni.

2.2 Vývojový cyklus

Při vypracovávání projektu jsme využívali verzovací systém Git. Pro dané části projektu byly vytvořeny konkrétní branche, kde jsme je testovali a upravovali. Před zahrnutím do master branche byl vyžadován pull request, následný code review a schválení od jednoho či více členů. Součástí vývojového cyklu byl i unit testing. Při každém commitu byly automaticky spuštěny testy a bylo tak hned možné vidět, jaký dopad bude commit mít na výsledný program.

3 Návrh a implementace překladače

Projekt byl rozdělen na několik konrétních částí, které jsou představeny v této kapitole.

3.1 Lexikální analýza

Prvním krokem překladače byl návrh a následná implementace lexikální analýzy. Celý scanner je implementován jako deterministický konečný automat, jehož diagram lze nalézt v obrázku 1.

Hlavní funkce tohoto tzv. scanneru je `getToken`, pomocí které se čte znak po znaku ze zdrojového souboru a převádí se na strukturu `Token`, která se skládá z typu, hodnoty a z důvodu vypisování chybových hlášek také pozice. Typy tokenu jsou `EOF`, prázdný token, identifikátory, klíčová slova, celé či desetinné číslo, řetězec a také porovnávací a aritmetické operátory a ostatní znaky, které mohou být použity v jazyce IFJ22. Hodnota tokenu je typu `union` a přiděluje se tokenům `TOKEN_KEYWORD`, `TOKEN_INT`, `TOKEN_FLOAT` a `TOKEN_STRING`.

Implementace se nachází ve zdrojovém souboru `scanner.c` a hlavičkovém souboru `scanner.h`. Jedná se o dlouhý `switch`, kde každý případ `case` je ekvivalentní k jednomu stavu automatu. Pokud načtený znak nesouhlasí s žádným znakem, který jazyk povoluje, program je ukončen a vrací chybu 1. Jinak se přechází do dalších stavů a načítají se další znaky, dokud nemáme hotový jeden token, tj. dokud nedostaneme `tokenComplete = true` a token potom vracíme a ukončíme tuto funkci.

Pro zpracovávání hexadecimálních a oktálních escape sekvencí v řetězci máme vytvořena dvě pole o velikosti 3 a 4, která jsou zpočátku vynulována a poté se postupně naplňují přečtenými čísly, vždy na pozici podle toho, kolikáté číslo zrovna uvažujeme a nakonec celé číslo převedeme do ASCII podoby.

3.1.1 Prolog a epilog

Na zpracování prologu jsme se rozhodli vytvořit funkci `initialScan`, která ve scanneru kontroluje první část, a to otevírací značku `<?php`, druhou část – příkaz aktivující přepínač: `declare(strict_types=1);`, posíláme jakožto jednotlivé tokeny dále syntaktické analýze, ve které provedeme dodatečnou kontrolu.

Pro zavírací značku volitelného epilogu je vytvořen `TOKEN_CLOSING_TAG`, za kterým je očekáván `EOF` a nebo `\n` a poté `EOF`, jinak vrací chybu 1.

3.1.2 Řetězec s proměnnou délkou

K zpracovávání identifikátorů a klíčových slov používáme vlastní strukturu `vStr`, která je implementována ve zdrojovém souboru `vstr.c` a souboru hlavičkovém `vstr.h`.

Jedná se o pole znaků, které se dynamicky zvětšuje v závislosti na své délce – při dosáhnutí aktuální maximální délky řetězce je délka zdvojnásobena. Znaky načítáme a průběžně ukládáme do dynamického pole a jakmile odcházíme ze stavu identifikátoru, porovnáváme, jestli načtený řetězec není shodný s nějakým z klíčových slov a podle toho se rozhodneme, zda vracíme typ a hodnotu tokenu jakožto klíčové slovo či identifikátor.

3.2 Syntaktická analýza

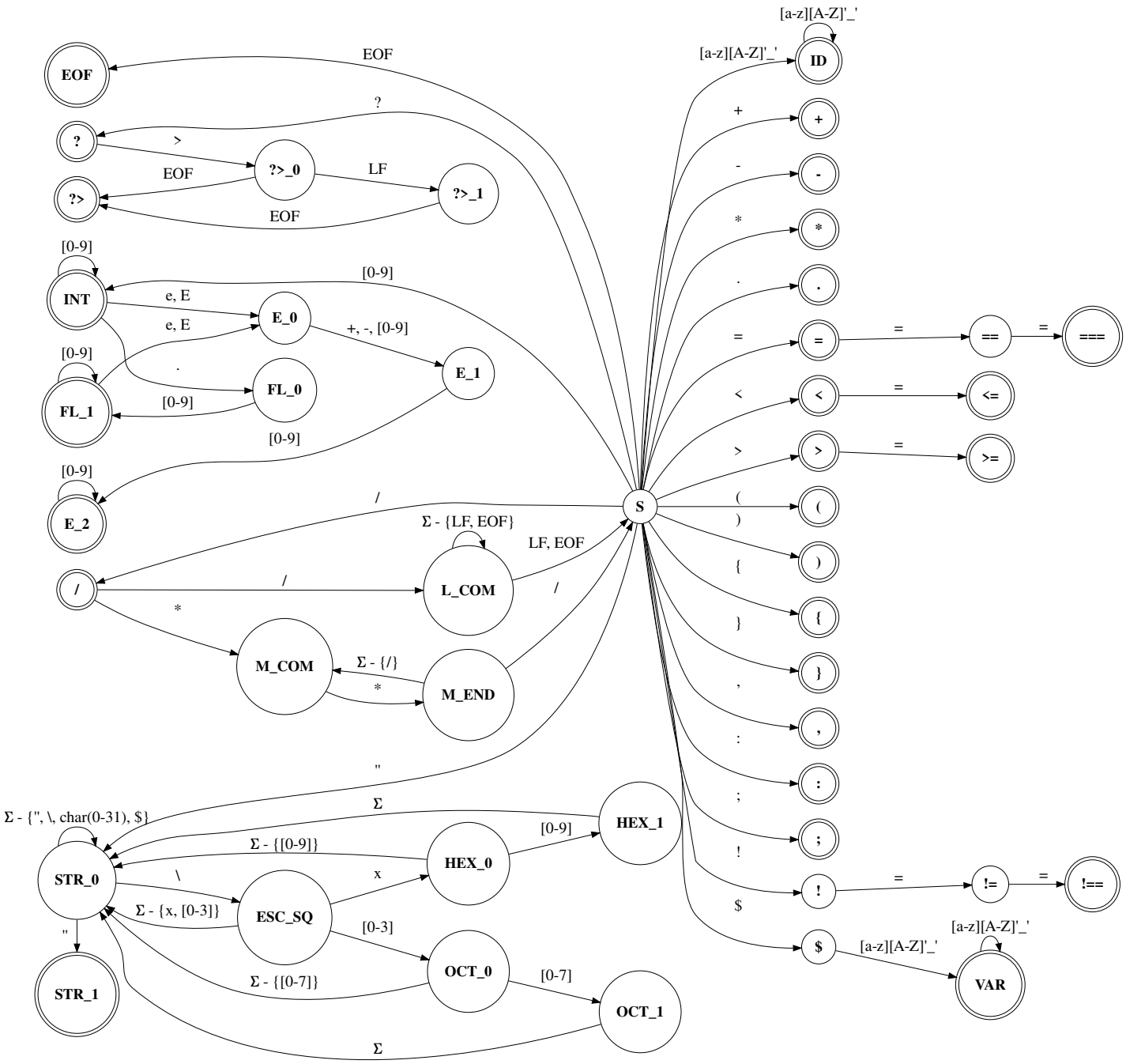
3.2.1 Precedenční syntaktická analýza

3.3 Sémantická analýza

3.4 Generování cílového kódu

3.5 Přílohy

3.5.1 Diagram konečného automatu



Obrázek 1: Konečný automat[1] specifikující lexikální analyzátor

[1]Z důvodu úspory místa a čitelnosti byly jednotlivé stavy přejmenovány, legenda stavů následuje na další stráně

Legenda

S	STATE_INITIAL	/	STATE_SLASH
ID	STATE_IDENTIFIER	L_COM	STATE_LINE_COM
+	STATE_PLUS	M_COM	STATE_MULTILINE_COM
-	STATE_MINUS	M_END	STATE_POT_MULTILINE_END
*	STATE_MULTIPLY	STR_0	STATE_STRING_0
.	STATE_CONCATENATE	STR_1	STATE_STRING_1
=	STATE_EQUAL_OR_ASSIGN	ESC_SQ	STATE_STRING_ESCAPE
==	STATE_EQUAL_0	HEX_0	STATE_STRING_HEXA_0
===	STATE_EQUAL_1	HEX_1	STATE_STRING_HEXA_1
<	STATE_LESSTHAN	OCT_0	STATE_STRING_OCTA_0
>	STATE_GREATERTHAN	OCT_1	STATE_STRING_OCTA_1
(	STATE_LEFT_BRACKET		
)	STATE_RIGHT_BRACKET		
{	STATE_LEFT_BRACE		
}	STATE_RIGHT_BRACE		
EOF	STATE_EOF		
?	STATE_OPTIONAL		
?>_0	STATE_CLOSING_TAG_0		
?>_1	STATE_CLOSING_TAG_1		
?>	STATE_CLOSING_TAG_2		
,	STATE_COMMA		
:	STATE_COLON		
;	STATE_SEMICOLON		
!	STATE_NOT_EQUAL_0		
!=	STATE_NOT_EQUAL_1		
!==	STATE_NOT_EQUAL_2		
$	STATE_VARID_PREFIX		
INT	STATE_NUMBER		
E_0	STATE_FLOAT_E_0		
E_1	STATE_FLOAT_E_1		
E_2	STATE_FLOAT_E_2		
FL_0	STATE_FLOAT_0		
FL_1	STATE_FLOAT_1		

3.5.2 LL – gramatika

1. <program> -> <body_main> <program_e>
2. <program_e> -> ε
3. <program_e> -> ?>
4. <type_p> -> float
5. <type_p> -> int
6. <type_p> -> string
7. <type> -> null
8. <type> -> <type_p>
9. <type> -> ?<type_p>
10. <return> -> return <return_p>
11. <return_p> -> expr
12. <return_p> -> ε
13. <body_main> -> <body>
14. <body_main> -> <func_def> <body>
15. <body> -> ε
16. <body> -> <if> ; <body>
17. <body> -> <while> ; <body>
18. <body> -> expr ; <body>
19. <body> -> identifier_var = expr ; <body>
20. <body> -> indentifier_var = <func> ; <body>
21. <body> -> <func> ; <body>
22. <body> -> <return> ; <body>
23. <if> -> if expr { <body> } else { <body> }
24. <while> -> while expr { <body> }
25. <params> -> ε
26. <params> -> <type> identifier_var <params_n>
27. <params> -> identifier_var <params_n>
28. <params> -> expr <params_n>
29. <params_n> -> ε
30. <params_n> -> , <params_p> <params_n>
31. <params_p> -> <type> identifier_var
32. <params_p> -> identifier_var
33. <params_p> -> expr
34. <func_def> -> function identifier_func (<params>) :
    <type> { <body> }
35. <func> -> identifier_func (<params>)

Tabulka 1: LL – gramatika řídící syntaktickou analýzu

3.5.3 LL – tabulka

3.5.4 Precedenční tabulka

4 Reference