VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace k projektu do IFJ/IAL

IMPLEMENTACE PŘEKLADAČE IMPERATIVNÍHO JAZYKA IFJ18

Tým 015, varianta I

Členové týmu:

David Gajdoš - xgajdo22 - 25%

Martin Macháček - xmacha73 - 25%

Ondřej Studnička - xstudn00 - 25%

Václav Trampeška - xtramp00 - 25%

5. prosince 2018

Obsah

1 Úvod

Cílem projektu bylo vytvořit program v jazyce C, který načítá zdrojový kód ve zdrojovém jazyce IFJ18 a poté ho překládá do cílového jazyka IFJcode18.

Program je konzolová aplikace, která načítá zdrojový soubor ze standardního vstupu a poté generuje mezikód na standardní výstup. V případě, že nastane chyba, volá funkci `error_exit_code`, která odpovídající chybový stav vypíše a ukončí program.

2 Návrh a implementace

Program je sestaven z několika vzájemně propojených modulů:

2.1 Lexikální analýza

Tvorba projektu začala návrhem a implementací lexikálního analyzátoru. Celý lexikální analyzátor je implementován jako deterministický konečný automat **(1)**. Hlavní část tohoto modulu je funkce `get_new_token`, která pomocí cyklu `while`, funkce `getc` a opakujícího se `switch` pročítá jednotlivé znaky ze vstupního souboru a převádí jej na strukturu Token porovnáváním načteného znaku se znaky dostupnými v aktuálním stavu automatu. Pokud se načtený znak nerovná žádnému z dostupných stavů, nastává chybový stav. Token má svůj typ a hodnotu.

Typ tokenu může být EOL, EOF, klíčové slovo, identifikátor, aritmetický nebo přiřazovací operátor, datový typ `string`, `integer` a `double`, čárka, dvojitá uvozovka, porovnávací operátor, mřížka (#) a nakonec kulaté závorky. Hodnota tokenu může být celé číslo, desetinné číslo, řetězec (buď jméno proměnné nebo samotný obsah řetězce), nebo jedna z klíčových slov.

Pro vyřešení blokových komentářů jsme použili proměnnou `is_first_token`, abychom určili, zda se může jednat o blokový komentář nebo ne. Návratová hodnota lexikální analýzy je v případě úspěšného dokončení 0, v opačném případě se jedná o lexikální chybu, tudíž je chybový stav nastaven na 1, dle zadání.

2.2. Syntaktická analýza

Syntaktická analýza, v souboru `parser.c`, je jedna z nejdůležitějších částí celého projektu, protože řídí chod programu jako celku pomocí předem vytvořených LL pravidel **(2)** a LL tabulky **(3)**.

Každé LL pravidlo je definováno jako funkce, ve které se kontroluje právě načtený token a zjišťují se další možnosti chodu programu. Jestli že se nenaskytne žádná další možnost chodu programu, nastane chybový stav 2.
Načítání nového tokenu se provádí voláním funkce `get_new_token` z lexikálního analyzátoru.

2.2.A Precedenční syntaktická analýza

Precedenční syntaktická analýza je další důležitou částí projektu, řeší se zde všechny výrazy pomocí tabulky precedenční syntaktické analýzy **(4)**. Algoritmus pro precedenční syntaktickou analýzu je inspirován algoritmem probíraným na přednášce předmětu IFJ **[2]** a je implementován ve funkci `psa`. Další hlavní funkcí je `get_new_intput`, kde je zpracován token a předán do funkce `psa`.

Do tohoto modulu přistupujeme ve chvíli, kdy nalezneme začátek výrazu, nebo když jistě víme, že bude výraz následovat. Tento modul je implementován v souboru `psa.c`, chybový stav výrazů je 4.

2.3. Sémantická analýza

Sémantická analýza postupně prochází symboly či skupiny symbolů ze syntaktické analýzy, přiřazuje jim datový typ a kontroluje, jestli již proměnná byla deklarována, aby se mohla použít. Do datové struktury `sym_table` jsou postupně vkládány uzly (proměnné) s informacemi o datovém typu, jestli se jedná o funkci, a jméno proměnné reprezentováno jako klíč daného uzlu.

Je zde využívána i precedenční syntaktická analýza z `psa.c`, kterou když propojíme se sémantickou analýzou, můžeme správně kontrolovat a měnit datové typy proměnných ve výrazech. Příklad nutnosti změny datového typu může být sečtení čísla `integer` a `double`, kde `integer` bude přetypován na `double`, protože kdybychom přetypovali `double` na `integer`, ztratili bychom hodnotu za desetinnou čárkou. Chybový stav sémantické analýzy je 3.

2.4. Generátor cílového kódu

Jedná se o generování kódu IFJcode18. Po zkontrolování sémantiky a syntaxe se volá generátor umístěný v souboru `generate_code.c`.

Nejdříve se inicializují potřebné datové struktury pro generování. Generátor funguje na principu instrukční pásky, kde se do jednosměrně vázaného seznamu `instr_list` nejprve načtou instrukce a následně se při průchodu od prvního prvku seznamu postupně generují a ukládají do pole řetězců.

Před vygenerováním samotné instrukce se vygenerují jednotlivé operandy, které vzápětí vygenerují danou instrukci. Instrukce se po vygenerování uloží na konec seznamu `instr_list`.

3 Komunikace mezi členy a práce v týmu

3.1 Zahájení a tvorba projektu

Projekt jsme začali vypracovávat v polovině listopadu. Ze začátku jsme si museli stanovit, na kolik a na jaké části bude projekt rozdělen, což nám trvalo docela dlouho, protože programu jako celku jsme ze začátku moc nerozuměli.

Začali jsme tedy určením verzovacího systému, způsobu komunikace a tvorbou lexikální analýzy, ze které jsme poté pokračovali dál na další moduly, které jsme většinou implementovali skupinově.

3.2 Verzovací systém

Jako verzovací systém jsme si zvolili Git, jako vzdálený repositář je použit GitHub, ve kterém jsme si vytvořili školní účty a nastavili oprávnění pro přístup k repositáři.

3.3 Komunikace mezi členy týmu

Komunikace mezi členy skupiny pro tvorbu projektu byla zajištěna prostřednictvím aplikace Discord, která poskytuje jak skupinový hovor se sdílením obrazovek, tak intuitivní prostředí pro komunikaci přes instant-messaging.

Během řešení projektu se uskutečnilo mnoho osobních setkání, kde jsme řešili problematiky jednotlivých modulů.

3.4 Rozdělení práce

Práce na projektu je rozdělena rovnoměrně, každému členu je tudíž přiřazeno procentuální hodnocení 25%. Tabulka níže shrnuje podílení jednotlivých členů na každé části projektu.

	Lexikální analýza	Syntaktická analýza	Sémantická analýza	Generování	Dokumentace	Testování	Prezentace
xgajdo22			X	X	X	X	
xmacha73	X	X			X	X	X
xstudn00		X		X	X	X	
xtramp00	X	X	X	X		X	

*: znak X v tabulce = ANO

4 Závěr

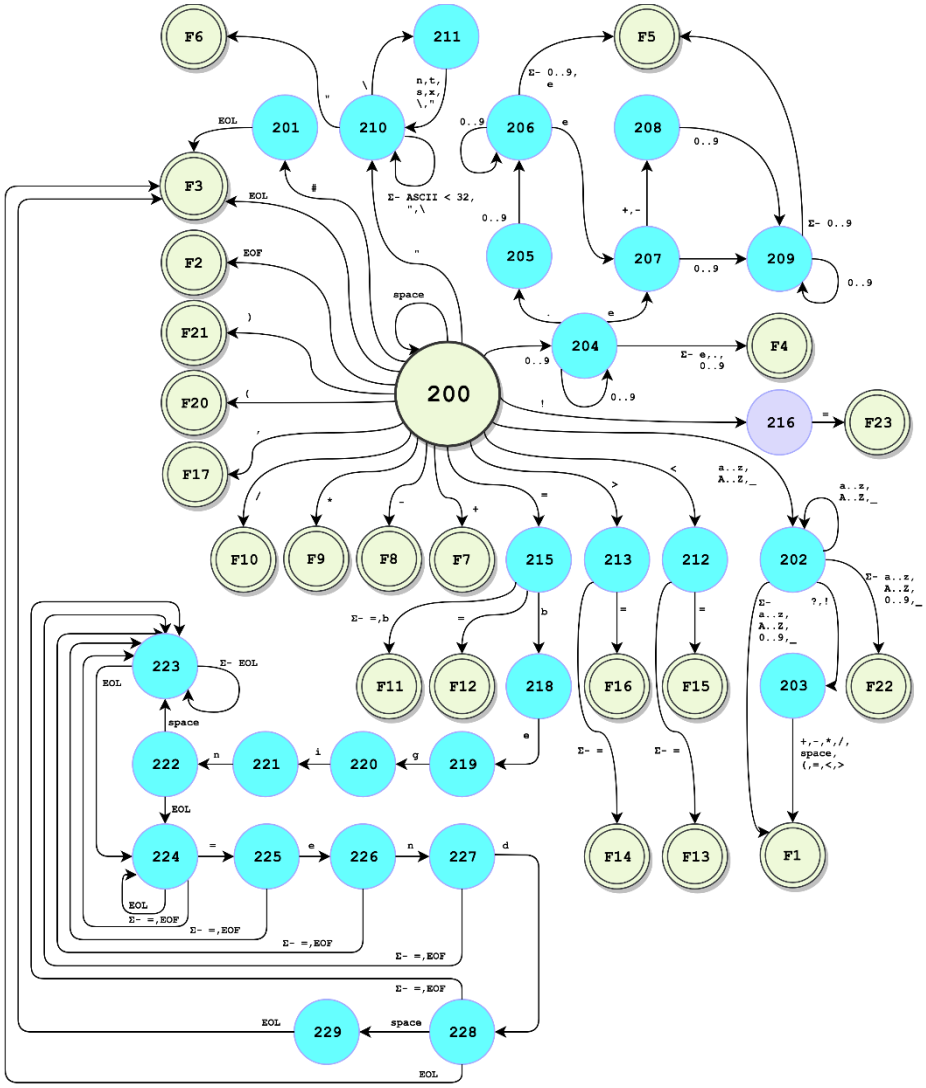
Zadání projektu nás z počátku zaskočilo, i přes to, že jsme od studentů z vyšších ročníků byli varováni předem o jeho náročnosti. Na přednáškách jsme sice získávali znalosti potřebné k řešení projektu, i přesto jsme neustále měli spoustu nejasností pro implementaci samotného řešení.

S projektem jsme začali pracovat trochu později než je doporučeno, takže jakmile jsme se dostali k části generování výsledného kódu, uvědomili jsme si, že by bylo mnohem jednodušší, kdybychom si například jednotlivé tokeny ukládali postupně do seznamu načtených tokenů, což by zjednodušilo řešení některých problémů, ale už by nám nezbývalo moc prostoru na testování, tudíž jsme pokračovali s původní implementací.

Vypracování projektů v předmětu IAL nás dobře připravilo na implementaci tabulky symbolů pro sémantickou analýzu, dále jsme hodně využívali návody a pomůcky pro vypracování projektu na webových stránkách k předmětu IFJ [1] a tohoto projektu.

Tento projekt nám přinesl spoustu znalostí a přehled o fungování překladačů a přinesl cenné zkušenosti s projekty většího rozsahu.

Diagram konečného automatu lexikální analýzy (1)



LEGENDA:

201	SCANNER_LINE_COMMENT
202	SCANNER_IDENTIFIER_OR_KEYWORD
203	SCANNER_QUESTION_EXCLAMATION
204	SCANNER_NUMBER
205	SCANNER_NUMBER_POINT
206	SCANNER_NUMBER_DOUBLE
207	SCANNER_NUMBER_EXPONENT
208	SCANNER_NUMBER_EXPONENT_SIGN
209	SCANNER_NUMBER_EXPONENT_LAST
210	SCANNER_STRING
211	SCANNER_STRING_ESCAPE
212	SCANNER_LESS_THAN
213	SCANNER_MORE_THAN
214	SCANNER_EOL
215	SCANNER_EQ_ASSIGN
216	SCANNER_NEQ
218	SCANNER_BLOCK_EQ
219	SCANNER_BLOCK_EQBE
220	SCANNER_BLOCK_EQBEG
221	SCANNER_BLOCK_EQBEGI
222	SCANNER_BLOCK_EQBEGIN
223	SCANNER_BLOCK_EQBEGIN_SPACE
224	SCANNER_BLOCK_INSIDE
225	SCANNER_BLOCK_INSIDE_OR_END
226	SCANNER_BLOCK_EQE
227	SCANNER_BLOCK_EQEN
228	SCANNER_BLOCK_EQEND
229	SCANNER_BLOCK_EQEND_MORE

F1	token_type_identifier
F2	token_type_eof
F3	token_type_eol
F4	token_type_int
F5	token_type_double
F6	token_type_string
F7	token_type_plus
F8	token_type_minus
F9	token_type_multiply
F10	token_type_divide
F11	token_type_assign
F12	token_type_eq
F13	token_type_less
F14	token_type_more
F15	token_type_leq
F16	token_type_meq
F17	token_type_comma
F20	token_type_obracket
F21	token_type_cbracket
F22	token_type_keword
F23	token_type_neq

LL – Gramatika (2)

1. <prog> -> <code>
2. <code> -> DEF ID (<params> <stat-list-end> EOL <code>
3. <code> -> <stat-list>
4. <stat-list> -> <stat> <code>
5. <stat-list> -> EOL <code>
6. <stat-list> -> EOF
7. <stat-list-else> -> <stat> <stat-list-else>
8. <stat-list-else> -> EOL <stat-list-else>
9. <stat-list-else> -> ELSE
10. <stat-list-end> -> <stat> <stat-list-end>
11. <stat-list-end> -> EOL <stat-list-end>
12. <stat-list-end> -> END
13. <stat> -> ID <assign-expr>
14. <stat> -> IF <expr> EOL <stat-list-else> EOL
    <stat-list-end> EOL
15. <stat> -> WHILE <expr> EOL <stat-list-end> EOL
16. <stat> -> (<expr>
17. <stat-> -> <item> <expr>
18. <assign-expr> -> = <expr>
19. <assign-expr> -> <expr>
20. <assign-expr> -> EOL
21. <params> ->) EOL
22. <params> -> ID <next-param>
23. <next-param> -> , ID <next-param>
24. <next-param> ->) EOL
25. <item> -> I_value
26. <item> -> S_value
27. <item> -> D_value
28. <item-> -> NIL

LL – Tabulka

(3)

	DEF	EOL	EOF	ELSE	END	ID	IF	WHILE	(	=	)	,	I_value	S_value	D_value	NIL
<prog>																
<code>	2															
<stat-list>		5	6													
<stat-list-else>		8		9												
<stat-list-end>		11			12											
<stat>						13	14	15	16							
<assign-expr>		20								18						
<params>						22					21					
<next-param>											24	23				
<item>													25	26	27	28

Tabulka precedenční analýzy

(4)

	*	/	+	-	<	>	>=	<=	==	!=	(	)	ID	$
*****	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	<	>	>	>	>	>	>	>	>	<	>	<	>
-	<	<	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	X	X	X	X	>	>	<	>	<	>
>	<	<	<	<	X	X	X	X	>	>	<	>	<	>
>=	<	<	<	<	X	X	X	X	>	>	<	>	<	>
<=	<	<	<	<	X	X	X	X	>	>	<	>	<	>
==	<	<	<	<	<	<	<	<	X	X	<	>	<	>
!=	<	<	<	<	<	<	<	<	X	X	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	X
)	>	>	>	>	>	>	>	>	>	>	X	>	X	>
ID	>	>	>	>	>	>	>	>	>	>	X	>	X	>
$	<	<	<	<	<	<	<	<	<	<	<	X	<	

Použité zdroje

[1] Doplňující informace k předmětu IFJ
https://www.fit.vutbr.cz/study/courses/IFJ/public/project

[2] Algoritmus precedenční syntaktické analýzy – Přednáška IFJ
Prezentace: ifj08-anim-cz.pdf ze stáhnutého archivu ifj-anim-pdf.zip
https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FIFJ-IT%2Flectures