Projektová dokumentace

Překladače pro staticky typovaný imperativní jazyk IFJ20

Tým 084, varianta II
Mikota Michal (xmikot01) 25%
Buráň Radim (xburan03) 25%
Mlčoch Michal (xmlcoc12) 25%
Oravec Dávid (xorave06) 25%

Rozšíření: UNARY a FUNEXP

9. prosince 2020

Obsah

1 Úvod

Cílem tohoto projektu bylo vytvořit překladač pro programovací jazyk IFJ20. Tento jazyk je podmnožinou jazyka Go a následně je přeložen do cílového jazyka IFJcode20. IFJcode20 představuje tříadresný mezikód.

Překladač je obsluhován pomocí konzole, který čte ze standardního vstupu zdrojový kód IFJ20 a na standardní výstup generuje mezikód IFJcode20. V případě jakékoliv chyby program vrací odpovídající návratovou hodnotu dle specifikace.

2 Řešení projektu

Projekt je sestaven z několika modulů, které budou níže popsány.

2.1 Lexikální analýza

V lexikální analýze je nejdůležitější si na začátku správně nadefinovat konečný deterministický automat (viz. diagram 2 str.9). Tento automat zpracovává znak po znaku, které jsou načítány ze standardního vstupu. Tímto automatem se řídí celá lexikální analýza.

Jednoduché stavy (stav, který zpracovává maximálně dva znaky za sebou) jsou reprezentovány v rámci většího příkazu `switch`. Stavy, které zpracovávají více než dva znaky mají své vlastní funkce, mezi tyto funkce patří stavy rozeznávající identifikátory, čísla, řetězce a komentáře.

Pro komunikaci se syntaktickou analýzou je vytvořena funkce `getToken`. Tato funkce přebírá strukturu `tokenizer`, která v sobě uchovává vrácený token, aktuálně zpracovávaný znak, jestli aktuální zank byl zpracován, errorový kód, `StringBuilder` strukturu a jestli nastal konec souboru (vstupu).

2.2 Syntaktická analýza

Syntaktickou analýzu řídí pravidla popsané v LL-gramatice (viz. 8 str. 10). Pravidla jsou implementované mocí rekurzivního průchodu ze současným sestrojování syntaktického stromu. Tento strom se vytváří na základě každého pravidla, které má definovanou strukturu ve funkci `PrimaryExpressionSyntax`. Při vytváření syntaktického stromu si syntaktická analýza od lexikálního analyzátoru žádá tokeny pomocí funkce `getToken` a následně kontroluje, jestli daný token je očekávaný. Syntaktická analýza též řeší prioritu operátorů na základě precedenční tabulky (viz. obr. 4 str.12) pomocí funkce `GetbBinOperatorPriority`,nebo `genUnOperatorPriority`, které vracejí prioritu daného operátoru.

Díky správné implementaci rekurzi se automaticky vytváří správně prioritizovaný syntaktický strom při zpracování výrazů. Syntaktická analýza končí správně, jak dostane každý očekávaný token a nesprávně pokud zjistí jakoukoli nesrovnalost na základě gramatiky.

Následně po správném ukončení syntaktické analýzy, je syntaktický strom celého programu dokončený a zpřístupněn sémantické analýze a generování cílového kódu.

2.3 Sémantická analýza

Sémantická analýza pracuje se syntaktickým stromem vytvořeným v syntaktické analýze. Prochází jej rekurzivně a z tohoto důvodu používá hlavně u výrazů strukturu `tExpReturnType`. Tato struktura obsahuje především chybový kód, díky kterému lze „probublat“ v rekurzi zpětně a dále datový typ a případně jeho hodnotu, pokud se jedná o konstantu. Díky tomuto zvládne sémantická analýza zjednodušit výrazy s konstantami.

Údaje o proměnných a definice funkcí jsou uloženy ve struktuře `tHashItem` v tabulkách s rozptýlenými položkami, které jsou spravovány ve struktuře `tScope`, kde je odkaz na globální tabulku s definicemi funkcí a dále vnořující se tabulky bloků s proměnnými, které tvoří jednosměnně vázaný list. Pokud se jedná o funkci, která ještě nebyla definovaná, tak se nastaví hodnota `declared` v `tHashItem`

na `false` a proběhne pokus o odhadnutí typů parametrů a návratových typů. Při definici funkce se zkontroluje, jestli funkce odpovídá.

Sémantika obsahuje 2 globální proměnné, `foundReturn`, která se stará o kontrolu návratových hodnot v `if/else` a `disableAssignement`, která zabraňuje změny konstant, pokud se nachází v části kódu, kde není předem jasná cesta (například `if/else`, `for cyklus`).

2.4 Generování cílového kódu

Generování cílového mezikódu IFJcode20 bylo úspěšně implementované v rámci našeho projektu. Kód je generován současně za běhu se sémantickou analýzou. V případě, že nenastane žádná sémantická chyba, dochází ke generovaní výstupu na standardní výstup. Generování mezikódu je taktéž obohacené o speciální komentáře, které podporují finální interpret.

2.4.1 Rozhraní generátoru mezikódu IFJcode20

Generátor je implementován jako samostatný soubor s názvem `gen_code.c`. V tomto souboru se nachází pomocné funkce pro generátor mezikódu. Tyto funkce, jako většina využitích datových struktur pro algoritmy jako např. jednosměrně vázaný seznam jsou uložené v hlavičkovém souboru `gen_code.h`. Všechny funkce jsou volané v souboru `semantics.c`, které se volají v určených sekcích.

2.4.2 Začátek generovaní mezikódu

Při spuštění souboru `gen_code.c` se inicializují všechny potřebné datové struktury. Taktéž se inicializují potřebné globální proměnné a využijí se elementární funkce, které musí vygenerovat každý spustitelný soubor. Následně potom je vygenerovaná funkcionalita vestavěných funkcí, které jsou implementovány na základě jejich definice v rámci zadání projektu.

2.4.3 Generování výrazů

Generování výrazů je pravděpodobně nejkomplexnější část celého generátoru mezikódu. Využívá syntaktický strom k přecházení jednotlivých větví a na základě priorit, generuje vhodné instrukce pro finální interpret. Taktéž jsou funkce, které tento strom prochází a které jsou využité v několika dalších funkcích generátoru, které jsou nezbytné.

2.4.4 Generátor funkcí

Generovaní funkcí je rozděleno pro vestavěné funkce a pro uživatelsky definované funkce. Jelikož přímá funkcionalita vestavěných funkcí je dopředu předepsána, jedná se o zjednodušenou funkcionalitu tj. není zapotřebí využívat také funkce pro generovaní mezikódu, jako je to při uživatelských funkcích. Do každé funkce, pomocí `Temporary Framu` vstupují všechny proměnné, které v ní mohou být využité včetně jejich parametrů. Taktéž jsou všechny návratové hodnoty uložené do vhodných proměnných pro danou funkci(pokud to funkce umožňuje).

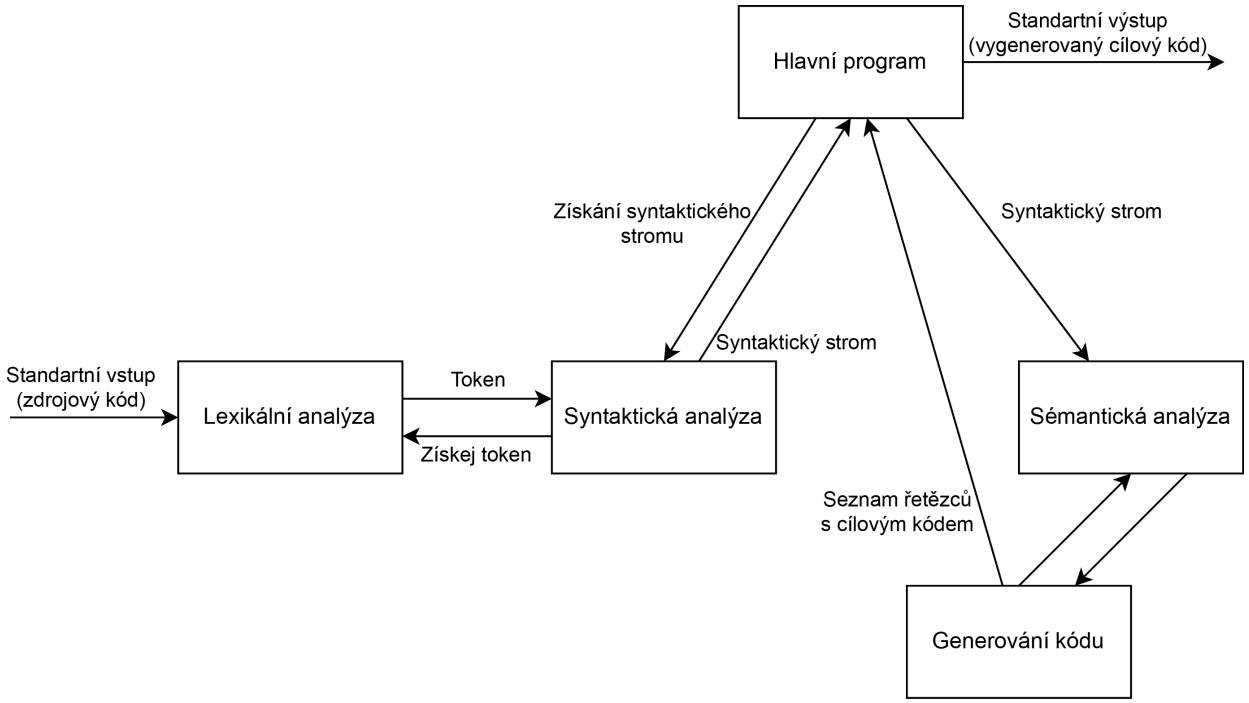
2.4.5 Generovaní návěští

Návěští jsou vytvářeny v co nejmenších počtech, aby se předešlo nepřehlednosti při generovaní. Zároveň jednotlivé návěští jsou pojmenovány tak, aby z nich bylo možné vyčíst, proč se tam nacházejí a v rámci kterých funkcí a za jakým účelem byly přidané. Pojmenování je zároveň vymyšleno tak, aby nedošlo k duplicitním pojmenováním, co by vedlo k interpretační chybě.

2.4.6 Závěr generování

V případě, že to bude možné (sémantika ani jiné části nevyhodí chybu) bude na standardní výstup vypsaný finální mezikód. Všechny instrukce a k nim přidané komentáře se nejprve uloží do samostatného listu na báze jednosměrně vázaného seznamu. Tento bude následně vypsaný na už vzpomínaný výstup. Samozřejmě, ať už budou jednotlivé části seznamu vypsané a nebo ne, bude seznam vhodně uvolněn.

2.5 Digram komunikace modulů



Obrázek 1: Digram komunikace modulů

3 Speciální algoritmy a datové struktury

3.1 Tabulka s rozptýlenými položkami

Při registraci varianty projektu jsme se rozhodli pro implementaci tabulky symbolů pomocí tabulky s rozptýlenými položkami. Tabulku jsme implementovali jako tabulku s explicitním zřetězováním synonym, aby jsme docílili toho, že tabulka bude omezena pouze velikostí dostupné paměti RAM. Pro zřetězování jsme použili jednostranně lineární vázaný seznam. Aby bylo dosaženo, co nejlepšího možného rozptýlení položek, používáme prvočísla jako velikost tabulky.

Jako mapovací funkci jsme použili FNV-1 hašovací funkci. Tato funkce je řazena jako ne-kryptografická hašovací funkce, proto tato funkce primárně slouží pro užití v tabulce s rozptýlenými položkami s tím souvisí i rychlost vypočítání konkrétní hodnoty. [1]

Každá položka obsahuje unikátní řetězec tzv. klíč, který je zároveň identifikátor proměnné, nebo funkce. Dále položka obsahuje ukazatel na další zřetězenou položku. Typ, který udává jakého charakteru je daný identifikátor jako: `int, string, func` apod. Pokud je typu funkce, potom položka obsahuje odkaz na strukturu, která v sobě uchovává parametry funkce, jejich typy, typy návratových hodnot a jejich počty. Jeli identifikátor proměnná, tak v sobě položka nese uchovanou hodnotu dané proměnné. V poslední řadě položka obsahuje pravdivostí hodnotu zda daný identifikátor již byl v minulosti deklarován.

K tomu aby vše se dalo jednoduše ovládat, tak jsme implementovali několik funkcí pro práci s tabulkou jako například: `inicializace`, `přidat položku`, `smazání položky`, `vyhledání položky` a `destruktor`.

3.2 Dynamicky vytvářený řetězec (tzv. String builder)

Programovací jazyk C v základu nemá žádnou funkci, která by dovolovala, přidávat nové znaky do již existujícího řetězce. Například tuto funkci mají již v základu vyšší programovací jazyky[1], proto je potřeba implementovat tuto funkci v C.

Tuto strukturu budeme potřebovat v lexikální části pro vytváření tokenů. Struktura obsahuje odkaz na aktuálně vytvářený řetězec, jeho délku a počet alokovaných bajtů. Hlavní implementovanou funkce je `appendChar` metoda, která přidává nový znak do řetězce, pokud není počet alokovaných bajtů dostatečný, funkce automaticky realokuje nový dostatečný prostor pro řetězec.

[1] Java, C# a další jiné programovací jazyky

4 Práce v týmu

4.1 Verzovací software

Jako verzovací software jsme využili Git. Pro jednoduchou zprávu repozitáře jsme zvolili GitHub pro jeho uložení.

Každý z modulů byl vytvářen ve vlastní větvi, které se následně do sebe slučovaly.

4.2 Komunikační kanály

Jako komunikační kanály jsme zvolil z počátku Messenger, pro jednoduché domlouvání. Následně byl použit Discord, který poskytuje pohodlnou komunikaci ať už chat, hovory nebo sdílení obrazu a mnoho dalších nástrojů. Jako jednu z užitečných funkcí jsme využili "GitHub Bot", který nás upozorňoval na jakékoliv změny v adresáři.

4.3 Rozdělení práce v týmu

Mikota Michal - generovaní cílového kódu

Buráň Radim - sémantická analýza

Mlčoch Michal - organizace práce, lexikální analýza

Oravec David - syntaktická analýza, generovaní cílového kódu

Dále se všichni podíleli na vypracování dokumentace, konzultacích a testování.

5 Závěr

Pro všechny byl tento projekt výzvou. Ze začátku bylo pro nás obtížné rozdělit sémantickou a syntaktickou analýzu. Po dlouhém studiu se nám podařilo pochopit koncept projektu a rozdělit projekt na části. Toto nás naučilo lépe komunikovat a navzájem spolupracovat. Zdokonalili jsme si práci s verzovacím systémem Git a s jeho technologickými vychytávky. Tento projekt nám přiblížil činnost překladačů, jeho strukturu a praktické využití algoritmů z předmětu IAL.

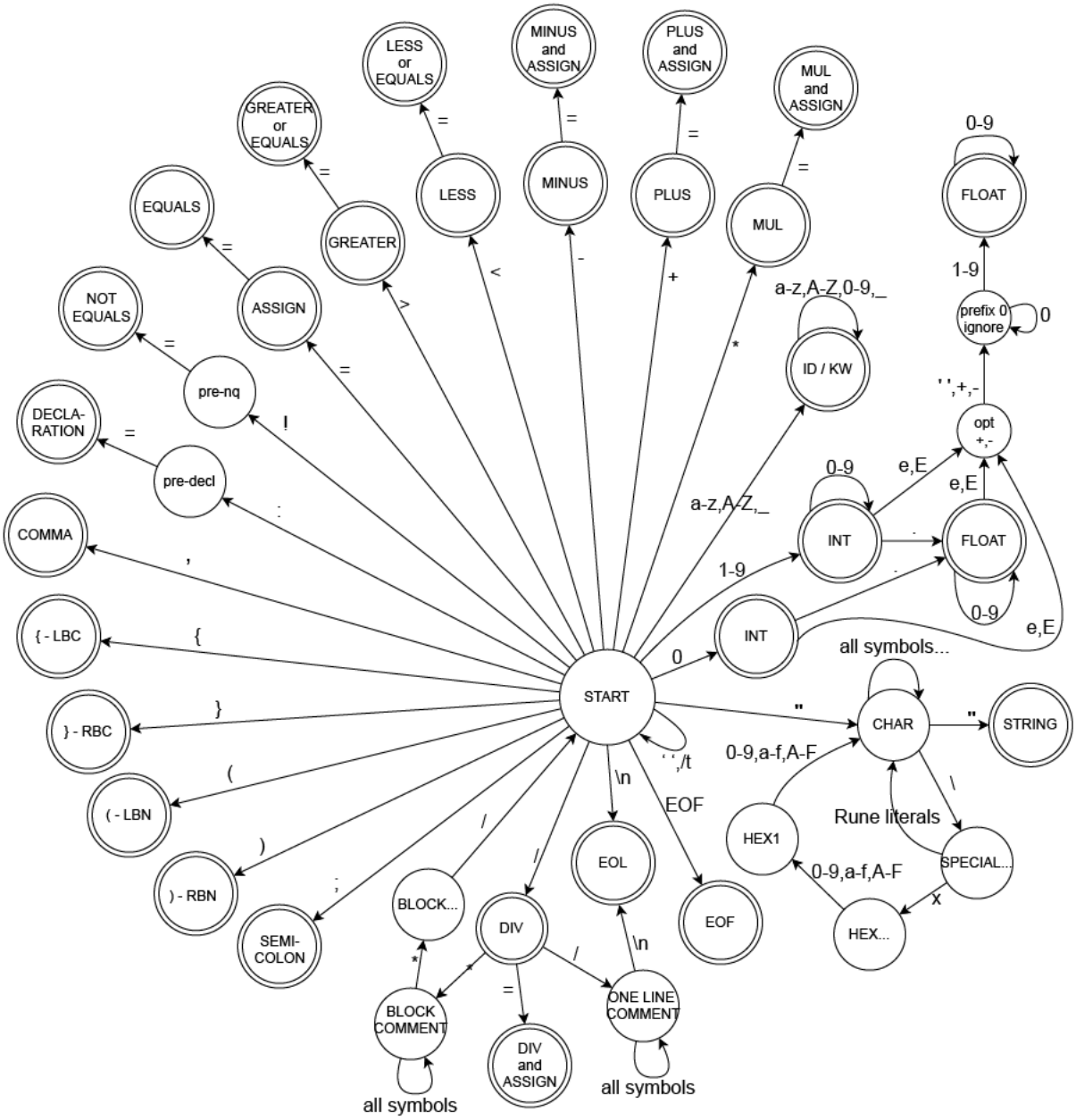
Doufáme, že náš překladač splnil všechny požadavky definované v rámci zadání tohoto projektu. Částečnou správnost našeho řešení jsme si ověřili nultým a prvním pokusným odevzdáním, díky čemu jsme zjistili, že jsme na správné cestě.

6 Literatura

Reference

[1] Fowler–Noll–Vo hash function [online]. [cit. 2020-12-03]. Dostupné z: https://en.wikipedia.org/wiki/Fowler-Noll-Vo_hash_function

7 Diagram konečného deterministického automatu lexikální analýzy



Obrázek 2: Diagram konečného deterministického automatu lexikální analýzy

8 LL - gramatika

1. prog -> package main EOL <prog>
2. prog -> func id(<param>) (<returnType>) { EOL <statement> } EOL <prog>
3. prog -> EOL <prog>
4. prog -> EOF

5. param   -> ID <type> <param_more>
6. param_more -> , ID <type> <param_more>
7. param   -> eps
8. param_more -> eps

9. returnType   -> <type> <returnType_more>
10. returnType_more -> , <type> <returnType_more>
11. returnType   -> eps
12. returnType_more -> eps

13. returnValue   -> <expr> <returnValue_more>
14. returnValue_more -> , <expr> <returnValue_more>
15. returnValue   -> eps
16. returnValue_more -> eps

17. type -> INT
18. type -> DOUBLE
19. type -> STRING

20. statement -> if <boolExpr*> { EOL <statements> }
                 else { EOL <statement> } EOL <statement>
21. statement -> for <decl>; <boolExpr*>;
                 <assign> { EOL <statement> } EOL <statement>
22. statement -> <decl>
23. statement -> <assign>
24. statement -> <assignUnary>
25. statement -> <expr>
26. statement -> eps
27. statement -> return <returnValue>

28. decl   -> ID := <expr> EOL <statement>
29. assign   -> ID, ID2, ... = expr, expr2, ...
30. assignUnary -> ID += <expr> EOL <statement>
31. assignUnary -> ID -= <expr> EOL <statement>
32. assignUnary -> ID *= <expr> EOL <statement>
33. assignUnary -> ID /= <expr> EOL <statement>

34. expr -> ID(<arg>)
35. expr -> <data>
36. expr -> <expr> + <expr>
37. expr -> <expr> - <expr>
38. expr -> <expr> * <expr>
39. expr -> <expr> / <expr>

40. `expr -> +<expr>`
41. `expr -> -<expr>`
42. `expr -> (<expr>)`
43. `expr -> <boolExpr>`

44. `data -> ID_VAL`
45. `data -> INT_VAL`
46. `data -> DOUBLE_VAL`
47. `data -> STRING_VAL`

48. `boolExpr -> <expr> >  <expr>`
49. `boolExpr -> <expr> >= <expr>`
50. `boolExpr -> <expr> <  <expr>`
51. `boolExpr -> <expr> <= <expr>`
52. `boolExpr -> <expr> == <expr>`
53. `boolExpr -> <expr> != <expr>`

54. `arg      -> <expr> <arg_more>`
55. `arg_more -> , <expr> <arg_more>`
56. `arg      -> eps`
57. `arg_more -> eps`

9 LL - tabulka

	package	func	EOL	EOF	ID	,	if	for	return	INT	DOUBLE	STRING	ID_VAL	INT_VAL	DOUBLE_VAL	STRING_VAL	$
<prog>	1	2	3	4													
<param>					5												6
<param_more>						7											8
<returnType>																	9,10
<returnType_more>						11											12
<returnValue>																	13,14
<returnValue_more>						15											16
<type>										17	18	19					
<statement>							20	21	27								22-26
<decl>					28												
<assign>					29												
<assignUnary>					30-33												
<expr>					34												35-43
<data>													44	45	46	47	
<boolExpr>																	48-53
<arg>																	54,55
<arg_more>						56											57

Obrázek 3: LL - tabulka

10 Precedenční tabulka

	+,-	*,/	(	)	i	$
+,-	>	<	<	>	<	>
*,/	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
i	>	>		>		>
$	<	<	<		<	<

Obrázek 4: Precedenční tabulka