Překladač jazyka IFJ2020

Formální jazyky a Překladače

Tým 114, varianta 2

Členové týmu,rozdělení práce a rozdělení bodů:

Jiří Hofírek (vedoucí)	- Dokumentace a lexikální analýza	- 8 %
Michal Řezník	- Parser, výrazy	- 25 %
Samuel Repka	- Sémantická a lexikální analýza	- 40 %
Timotej Kamenský	- Lexikální analýza, pomocné funkce	- 27 %

Práce v týmu

VSC

Jako verzovací software jsme použili Git a Github, kteří nám byli užitečnými pomocníky při tvorbě projektu. Někdy jsme se však dostali až do stavu, kdy byly ve třech větvích 3 nedodělané části, které nikdo nespojoval. Chyběl někdo, kdo by na to dohlídl.

Komunikace

Komunikace byla značně ztížena tím, že ze zřejmých důvodů mohla probíhat pouze online. Komunikovali jsme tedy prostřednictvím aplikace discord, nebo telefonicky. Jako velice užitečná se ukázala možnost sdílení obrazovky.

Přesto toto omezení přispělo k drobným nedorozuměním. A dokonce se nám stalo, že v jednu chvíli neplánovaně pracovali dva členové týmu současně na stejné funkcionalitě každý ve vlastním souboru.

Problémem bylo také plnění deadlinů, nakolik hlavně z počátku chyběla struktura naší organizace.

Problémy v komunikaci a časovém manažmentu vyvrcholily 2 dny před posledním pokusným odevzdáním, kdy jsme zjistili, že Jiří Hofírek ani náhodou nestíhá dodělat scanner, a musel to za něj přebrat další člen týmu. Jeho porovnatelně nízké hodnocení je důsledkem taky tohoto incidentu.

Rozdelení práce

Práce na projektu ve výsledku nebyla rovnocenná. Zvolená architektura, kdy se generování kódu prolíná se syntaktickou a sémantickou analýzou způsobila vyšší zátěž na Samuela Repku, zatímco Jiří Hofírek kromě jiných nedokázal udělat scanner. Výsledné rozdělení práce je následovné:

Člen týmu	Přidělená práce
Samuel Repka	De-facto leader týmu, syntaktická a sémantická analýza, část generování kódu, část pomocných funkcí
Timotej Kamenský	Scanner, pomocné datové struktury, část dokumentace
Michal Řezník	Parser a zpracování výrazů
Jiří Hofírek	Dokumentace, pomocné funkce, část scanneru

Lexikální analýza

Průběh práce

Soubor s lexikální analízou scanner.c, scanner.h jako první začal programovat Jiří. Po dokončení těchto souborů nebyl Jiří schopen se zbavit se chyby segmentation fault po vykonání programu.

Jelikož se mu nepodařilo soubory zdebugovat, Timotej vytvořil soubory newscaner.c a newscaner.h.Kde začal programovat lexikální analízu od znovu. V těchto souborech s drobnou pomocí Jiřího (se závěrečným debuggingem pomáhal i Samuel) dokončil.

Implementace

Tento nový scanner je implementován jako Deterministický Konečný Automat. Avšak, v mnohých místech bylo v zmysle zjednodušení modelu nutné “nakouknout” na následující znak. Tento znak jsme si zachovali a znovu použili u dalšího cyklu automatu pomocí série příkazů “isbuff = true, buffedchar = c”. Tento buffer jsme pak použili i pro uložení znaku při zpracovávání hexadecimálního čísla, jelikož už byl tak trochu po ruce.

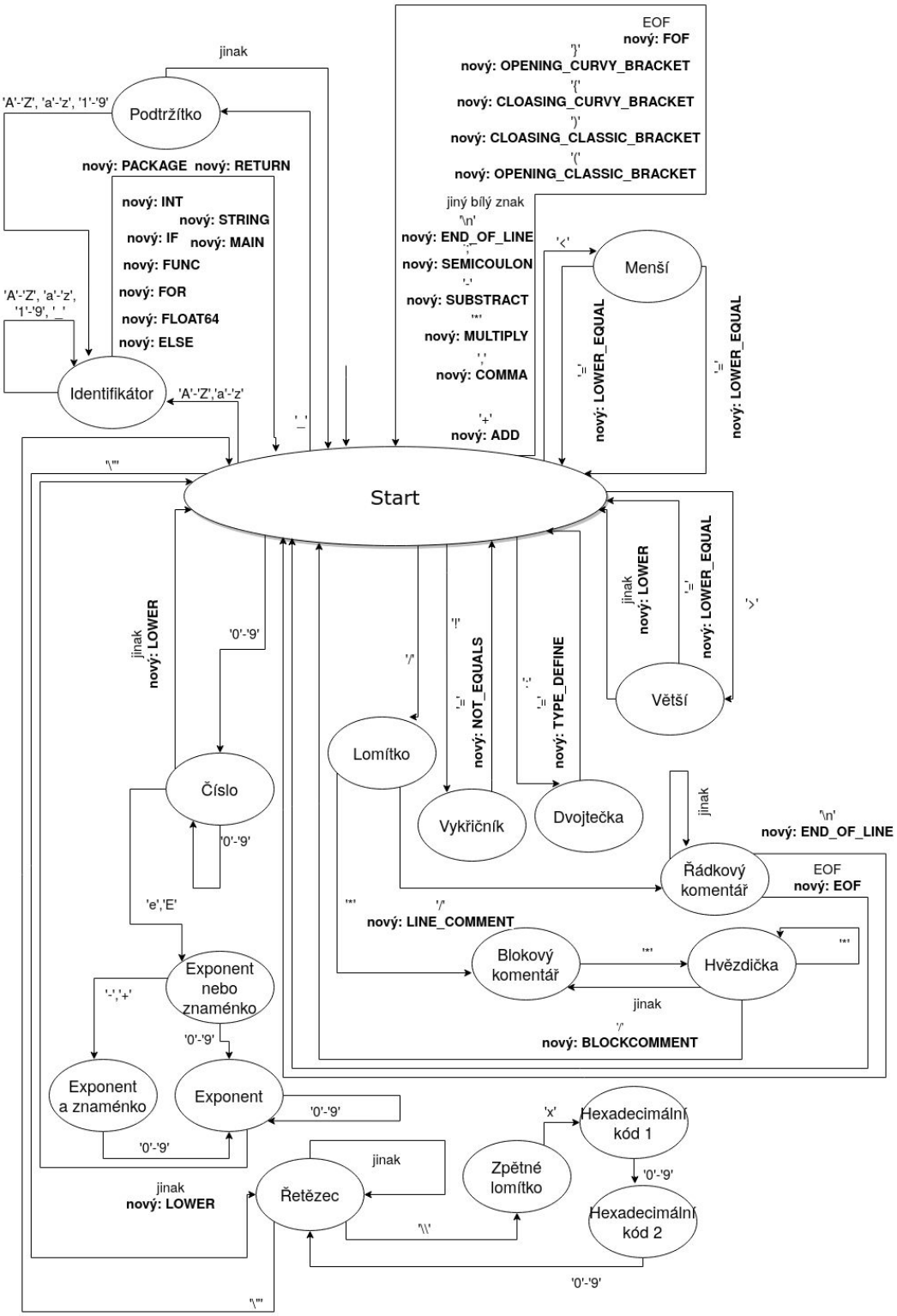
Tento automat má všechny své stavy uloženy jako ENUM v souboru newscanner.h.

Automat měl původně za úkol jenom naplnit v parametru posunutý token novými údaji. V pozdější fázi projektu se však ukázalo, že všechny tokeny bude potřeba projít dvakrát. K tomuto účelu sme “retrofitovali” už existující a docela dobře otestovaný scanner, a to tak, aby naplněný token ještě před návratem okopíroval a uložil do seznamu tokenů, který byl taky doráběn až dodatečně.

Taktéž užíváme doplňující datovou strukturu Dynamic_string, která funguje jako buffer řetězců v rámci scanneru a taky je obsahem tokenů typů String a identifier. Struktura drží char * a taky metadata o něm. To nám umožňuje aplikovat realloc vždy, kdy je string plný, a to samostatně u přidávání písmena nebo tokenu.

Diagram

Uzly grafu jsou popsány názvem stavu, ve kterém se automat nachází. Namísto klasické varianty zapsání přidání tokenu podle koncového stavu, je zde volena varianta zapsat přidání tokenu k hraně, při jejímž přechodu se token přidává (tučně „nový:“). Pro vyšší přehlednost je může být více hran znázorněných jako jediná (např. selfloop hrana uzlu start).



Syntaktická analýza

Průběh práce

Od začátku bylo jasně určeno, že celkové vyhotovení syntaktické analýzy bude mít na starosti Samuel, zatímco precedenční analýzu bude mít na starosti Michal. Bohužel se nám kvůli průtahům u lexikální analýzy nepovedlo zprovoznit syntaktickou analýzu ani do druhého pokusného odevzdání. Vývoj tak postupoval tak trochu naslepo.

Aby nedošlo k problémům mezi lexikální a syntaktickou analýzou, ještě před ukončením kterékoliv části bylo dohodnuto, které typy tokenů je a které není potřeba, a byl průběžně aktualizován. Takovým způsobem byl třeba dodatečně přidán token type main.

Dalším problémem se ukázalo být nedostatek výučbových materiálů. Přednášky jsme častokrát z rodinných nebo i jiných důvodů nemohli vidět v živém přenosu a respektovali jsme nařízení o nenahrávání přednášek spolu se slibem, že budou obratem (cca do týdne) zveřejňovány na videoserveru FIT. Zveřejněny však byli až krátce před celkovým odevzdáním (6.12. Ve večerních hodinách). Tato skutečnost mi překáží a nerozumím, proč tomu tak je.

Taktéž precedenční analýza se nám zdala jako docela náročný koncept k pochopení. V spojení s chybějícími materiály to pro nás znamenalo hledání přednášek z předešlých let, nebo sledování obskurních návodů na platformě Youtube, z velké části s hodně špatnou angličtinou. Nakonec se to ale Michalovi povedlo dotlačit do zdárného konce.

Implementace výrazů

Zpracování výrazů probíhá tehdy, kdy jej dle syntaxe očekáváme. Je spouštěna voláním StartExpr. To následně pomocí precedenční tabulky a dalších pomocných volání nalezne relaci.K práci s tabulkou jsme použili zásobník(který se, podobně Dynamic_stringu, automaticky prohlubuje). Obsahem zásobníku jsou struktury item obsahující informace o dané položce.

Při hlubším pohledu na volání uvnitř volání start expr vidíme, že nejdřív inicializujeme zásobník itemů. Voláme tokeny a základě precedenční tabulky projdeme celý výraz. Jestliže jde o otevřenou relaci, vložíme token do zásobníku. V opačném případě dále voláme WhileR, který má za úkol pracovat s hodnotami v zásobníku, hledat chyby datových typů a syntaxe.

Kvůli práci s proměnnými jsme použili další abstraktní datový typ - zásobník tabulek. Tabulky byly samozřejmě vytvořené dle zadání jako tabulky s rozptýlenými prvky. Na nich jsme však vytvořili automaticky se prohlubující zásobník tabulek, který je implementovaný jako list ukazatelů na ně.

Pomocí větvení nakonec rozhodujeme o výsledném typu funkce, a ihned tlačíme kód v jazyce IFJcode20. Výsledkem je tak fakt, že někdy se najdou chyby, i když jsme už tlačili část kódu. Není to elegantné, no je to funkčné řešení situace.

Zpracování výrazu končí s vyprázdněním stacku.

Precedenčná tabulka

	+	-	*	/	<	>	<=	>=	==	!=	(	)	val	$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<					>	>	<	>	<	>
>	<	<	<	<					>	>	<	>	<	>
<=	<	<	<	<					>	>	<	>	<	>
>=	<	<	<	<					>	>	<	>	<	>
==	<	<	<	<	<	<	<	<			<	>	<	>
!=	<	<	<	<	<	<	<	<			<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
val	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<		<	

Implementace syntaktiky

Syntaktika je implementována rekurzivním sestupem pomocí LL tabulky.. Zajímavým problémem a zároveň výzvou se ukázalo být přiřazení více údajú pomoci jednoho znaku přiřazení. Vyřešilo se to pomocí rozdílných stavů pro přiřazení jednoho a více proměnných.

V této části projektu jsme také změnili původně předpokládaný jednosměrně vázaný list tabulek na už vzpomínán automaticky se prohlubující zásobník. Ukázalo se totiž, že budem potřebovat jet mezi tabulkami i opačným směrem, a mezi obousměrně vázaným listem a zásobníkem pointerů jsme si vybrali právě druhé řešení.

Dalším problémem bylo vyřešit přiřazení hodnoty prvku jako funkční hodnotu a jako konstantu. Opět jsme užili neelegantního řešení a vytvořili dva různé stavy pro dvě různé situace.

LL - tabulka

Column1	if	else	for	func	return	package	float 64	int	string	main	id	(	)	{	}	,	#ERROR!	:=	;	eol	eof	expr
<prolog>				1																		
<f_list>				2																	3	
<func>										5	4											
<f_init>												6										
<f_call>												7										
<body>														8								
<param_def_list>											9		10									
<ret_t_list>							13	13	13				14									
<ret_t_list_n>													16			15						
<stat>	17		18		19						20				21							
<stat_list>	22		22		22						22				23							
<if>	24																					
<else>	26	25	26		26						26											
<for>			27																			
<return>					28																	
<expr_list>																				31		30
<expr_list_n>																50				51		
<id_n>												32				34	34	33				
<id_def>																		36				
<id_list>																39						
<id_list_n>																40	41					
<id_assign>																41	42					
<param>																						44
<param_list>																						45
<param_list_n>													47			46						
<eols>	49		49		49						49		49	49						48	49	

LL gramatika

1. <prolog> -> <eols> package main <eol> <eols> <f_list> <eols> <eof>

---

2. <f_list> -> func <func> <eols> <f_list>
3. <f_list> -> e

---

4. <func> -> id <f_init> <body>
5. <func> -> main <f_init> <body>

---

6. <f_init> -> (<param_def_list>) <ret_t_list>

---

7. <body> -> {<eol> <eols> <stat_list>}

---

8. <param_def_list> -> id <type> <param_def_list_n>
9. <param_def_list> -> e

---

10. <param_def_list_n> -> , <eols> id <type> <param_def_list_n>
11. <param_def_list_n> -> e

---

12. <ret_t_list> -> (<type> <ret_t_list_n>)
13. <ret_t_list> -> e

---

14. <ret_t_list_n> -> , <type> <ret_t_list_n>
15. <ret_t_list_n> -> e

---

16. <stat> -> <if>
17. <stat> -> <for>
18. <stat> -> <return>
19. <stat> -> id <id_n>
20. <stat> -> e
21. DELETED

---

22. <stat_list> -> <stat> <eol> <eols> <stat_list>
23. <stat_list> -> e

---

24. <if> -> if <expr><body><else>

---

25. <else> -> else <body>
26. <else> -> e

---

27. <for> -> for <id_def_v>; <expr>; <id_assign_v> <body>

---

28. <return> -> return <expr_list>

---

29. <expr>

---

30. <expr_list> -> <expr> <expr_list_n>
31. <expr_list> -> e

---
32. <expr_list_n> -> ,<expr> <expr_list_n>
33. <expr_list_n> -> e
---
34. <id_n> -> := <id_def>
35. <id_n> -> = <id_assign>
36. <id_n> -> , <id_list_assign>
37. <id_n> -> (<f_call>)
---
38. <id_def> -> <expr>
---
39. <id_def_v> -> <id_def>
40. <id_def_v> -> e
---
41. <id_assign> -> <expr>
---
42. <id_list_assign> -> , <id_list> = <expr_list>
42.2<id_list_assign> -> , <id_list> = <func_assign> <f_call>
---
43. <id_list> -> id <id_list_n>
---
44. <id_list_n> -> ,id <id_list_n>
45. <id_list_n> -> e
---
46. <f_call> -> <param_list>
---
47. <param_list> -> <expr> <param_list_n>
---
48. <param_list_n> -> , <expr> <param_list_n>
49. <param_list_n> -> e
50. DELETED
51. <eols> -> eol <eols>
52. <eols> -> e
---
53. <type> -> int
54. <type> -> float64
55. <type> -> string
---
56. <id_assign_v> -> <expr>
57. <id_assign_v> -> e
---
58. <func_assign>
---
59. <expr_list_assign> -> <expr> <expr_list_assign_n>
60. <expr_list_assign_n> -> <expr> <expr_list_assign_n>

Sémantická analýza a generování kódu

Sémantická analýza se od syntaktické analýzy nedá jednoduše oddělit, jsou totiž hodně protkané, avšak stále dochází k posunu dat mezi nimi u volání (třeba v porovnaní se scannerem, kterému jen podávame parametry tokenu a tokenlistu a jinak jde o plně samostatnou entitu). Generování kódu také probíha jak u zpracování výrazů, tak v sémantické analýze. Proto jsou hranice mezi jednotlivými komponentami v rozdělení generování kódu poněkud mlžité.

Definice vestavěných funkcíi jsou umístěna v souboru code_generator.c.