

Vysoké učení technické v Brně

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

IFJ PROJEKT

Projektová dokumentace

Tým xmesikj00, varianta **vv-BVS**

rozdelenie:

xkinzea00 20%

xkrejce00 25%

xholbin00 35%

xmesikj00 20%

členovia tímu:

Adam Kinzel xkinzea00

Eliška Krejčíková xkrejce00

Natália Holbíková xholbin00

Juraj Mesík **xmesikj00**

rozšírenia: FUNEXP, BOOLTHEN

Obsah

[Obsah](#)

[Rozdělení práce](#)

[Lexikální analýza](#)

[Syntaktická analýza](#)

[Precedenční analýza](#)

[Sémantická analýza](#)

[Tabulka symbolů](#)

[AST](#)

[Generátor](#)

Rozdělení práce

xmesikj00:

precedenční analýza - psa.*; expr_stack.*; precedence_table.*

lexikální analýza - scanner.*; error.*

syntaktická analýza - ast.*

xkinzea00:

syntaktická analýza - parser.*; ast.*

xkrejce00:

syntaktická analýza - parser.*

sémantická analýza - semantics.*

tabulka symbolů - symtable.*

xholbin00:

precedenční analýza - psa.* (rozšíření FUNEXP); expr_stack.*

lexikální analýza - scanner.*

syntaktická analýza - parser.*; ast.*

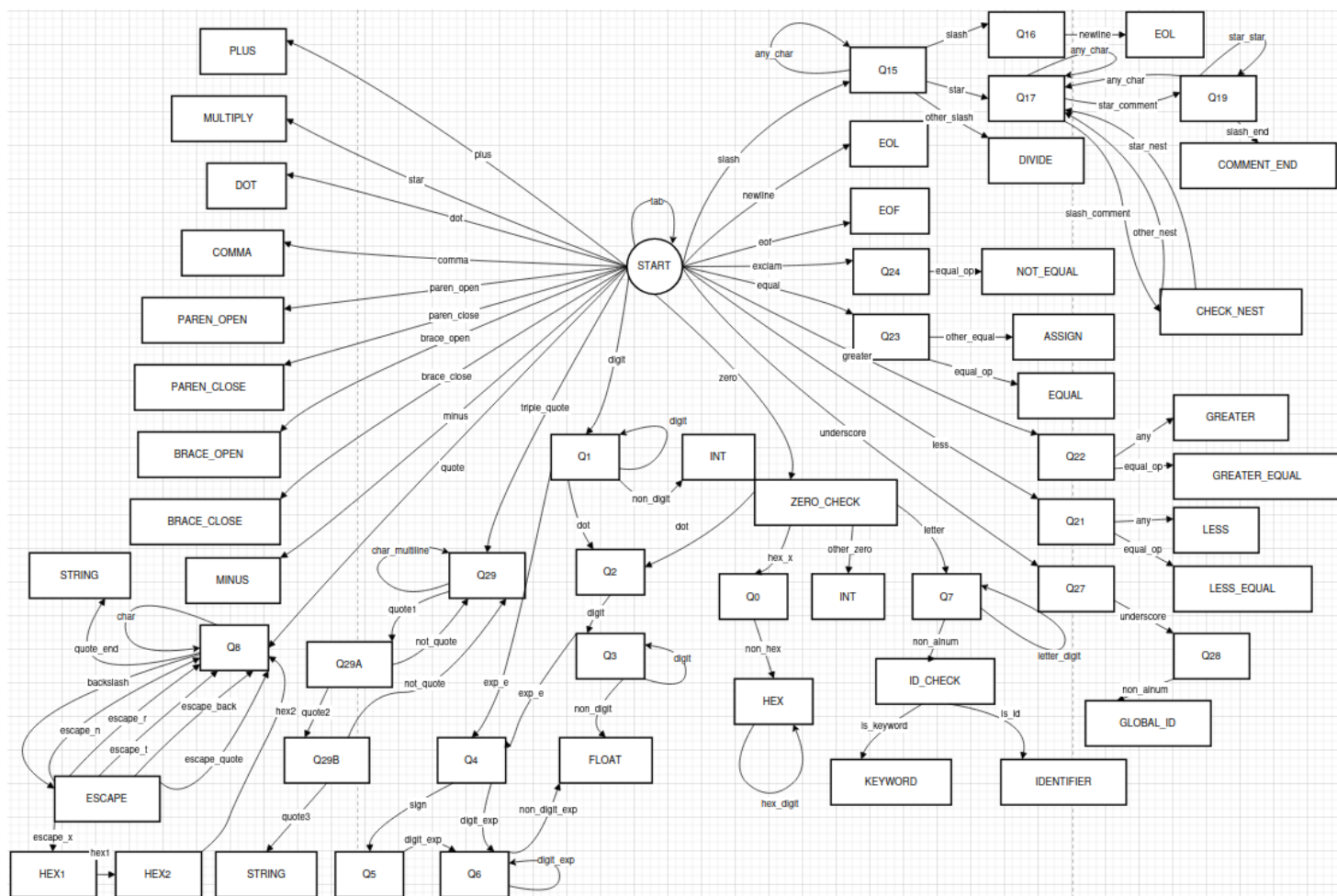
generovanie kódu - generator.*

testy

Lexikální analýza

[*scanner.**]

Lexikálnu analýzu sme implementovali ako konečný automat, ktorý podľa zadania ignoruje vybrané biele znaky a pomocou *createToken()* funkcie vytvára zo vstupu v jazyku IFJ25 relevantné tokeny, ktoré sú neskôr použité v ostatných častiach prekladača. Funkcia *getToken()* je zodpovedná za vrátenie štruktúry nasledujúceho tokenu zo štandardného vstupu.



obr. 1: konečný automat

Syntaktická analýza

Syntaktickou analýzu jsme implementovali jako jednorůchodový rekurzivní sestup. Jedná se o skupinu funkcí, která odpovídá jednotlivým pravidlům gramatiky.

Postupně se jednotlivé funkce volají a využíváme funkci `getTokenSyntax()`, která získává token za pomoci funkce `getToken()` a zároveň filtruje komentáře. U jednotlivých tokenů jsme následně kontrolovali typ, často za pomoci funkcí `check_token()` a `check_token_by_str_value()`. Jestli některý token neodpovídá pravidlům gramatiky, program je ukončen syntaktickou chybou.

Během průchodu zároveň vytváříme `AstNode`, přidáváme jej do AST a provádíme sémantickou kontrolu.

LL gramatika:

- (1) Program \rightarrow Prolog ClassDefinition EOF
- (2) Prolog \rightarrow import EolRuleNext "ifj25" EolRuleNext for EolRuleNext Ifj EolRule
- (3) ClassDefinition \rightarrow class Program { EolRule FunctionList }
- (4) FunctionList \rightarrow FunctionDef FunctionList
- (5) FunctionList $\rightarrow \epsilon$
- (6) FunctionDef \rightarrow static FunctionHeader Block EolRule
- (7) FunctionHeader \rightarrow identifier FunctionHeaderTail
- (8) FunctionHeaderTail \rightarrow (ParameterList)
- (9) FunctionHeaderTail \rightarrow = EolRuleNext (identifier)
- (10) FunctionHeaderTail $\rightarrow \epsilon$
- (11) ParameterList \rightarrow identifier ParameterListTail
- (12) ParameterList $\rightarrow \epsilon$
- (13) ParameterListTail \rightarrow , EolRuleNext identifier ParameterListTail
- (14) ParameterListTail $\rightarrow \epsilon$
- (15) Block \rightarrow { EolRule StatementList }
- (16) StatementList \rightarrow Statement StatementList
- (17) StatementList $\rightarrow \epsilon$
- (18) Statement \rightarrow var identifier EolRule
- (19) Statement \rightarrow if (Expression) Block IfElse
- (20) Statement \rightarrow while (Expression) Block EolRule
- (21) Statement \rightarrow return Expression EolRule
- (22) Statement \rightarrow Block
- (23) Statement \rightarrow identifier StatementTail
- (24) Statement \rightarrow global_identifier = EolRuleNext Expression EolRule
- (25) Statement \rightarrow Ifj . EolRuleNext identifier (ArgumentList) EolRule
- (26) StatementTail \rightarrow = EolRuleNext Expression EolRule
- (27) StatementTail \rightarrow (ArgumentList) EolRule
- (28) ArgumentList \rightarrow Term ArgumentListTail
- (29) ArgumentList $\rightarrow \epsilon$
- (30) ArgumentListTail \rightarrow , EolRuleNext Term ArgumentListTail
- (31) ArgumentListTail $\rightarrow \epsilon$
- (32) Term \rightarrow identifier
- (33) Term \rightarrow global_identifier
- (34) Term \rightarrow Literal
- (35) Literal \rightarrow int_literal
- (36) Literal \rightarrow float_literal
- (37) Literal \rightarrow string_literal
- (38) Literal \rightarrow hex_literal
- (39) Literal \rightarrow null
- (40) EolRule \rightarrow EOL EolRuleNext
- (41) EolRuleNext \rightarrow EOL EolRuleNext
- (42) EolRuleNext $\rightarrow \epsilon$
- (43) IfElse \rightarrow else Block EolRule
- (44) IfElse \rightarrow EolRule
- (45) Expression \rightarrow expression

LL tabulka:

	#	EXP	15 TO EXP_1	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895	896	897	898	899	900	901	902	903	904	905	906	907	908	909	910	911	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927	928	929	930	931	932	933	934	935	936	937	938	939	940	941	942	943	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975	976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991	992	993	994	995	996	997	998	999	1000

obr. 2: LL tabulka

Precedenční analýza

[psa.*, expr_stack.*, prec_table.*]

Precedenčná analýza má na starosti vyhodnocovanie výrazov podľa priorít uvedených v precedenčnej tabuľke (obr. 2). Implementovaná je pomocou analýzy zdola nahor s použitím zásobníka, čo v praxi znamená, že ako prvé sa vyhodnocujú operandy na najnižšej úrovni (identifikátory, literaly rôznych typov) a neskôr sa vyhodnotia a zredukujú operácie už vyhodnotených operandov (neterminálov), tak aby na konci analýzy zostal na zásobníku jediný neterminál. Tento prístup nám umožnil efektívne odchytiť všetky zakázané konštrukcie vo výrazoch. Hlavná funkcionality na implementovanú vo funkcii *parseExpr()*, ktorej sú predané potrebné parametre, a to hlavne štruktúra zásobníku, aktuálny token, očakávaná operácia, rodičovský uzol AST a zásobník tabuľky symbolov.

Počas implementácie sme si všimli, že v jazyku Wren musia byť podmienky v *if* a *while* konštrukciách zabalené v zátvorkách. Keďže precedenčná analýza je v týchto prípadoch podľa našej LL tabuľky zavolaná až po otváracíj zátvorke. V pôvodnej verzii by sa správne vyhodnotil výraz, ale na zásobníku by ostal jediný terminál zatváracíj zátvorky, čo by viedlo k chybovej hláške pri správnom vyraze. Museli sme teda nájsť spôsob ako odlišiť, či sa jedná o zatváraciu zátvorku, ktorá je súčasťou výrazu, alebo nie. Toto sme dosiahli zavedením novej operácie v precedenčnej tabuľke - *EOE* (*značka* 'o'), a ukladaním posledného tokenu pri každom posune. Takže v prípade, že je na zásobníku iba zatváracia zátvorka, znamená to, že sme narazili na koniec definície podmienky, a teda sa nejedná o token, ktorý je súčasťou precedenčnej analýzy - v takom prípade vrátíme do hlavného programu predošlý token, ktorý sme si ukladali práve kvôli tomuto prípadu.

		INPUT																			
		+	-	*	/	()	<	>	<=	>=	==	!=	\$	Num	String	null	Identif	is	keyword	
TOTOP OF STACK	+	>	>	<	<	<		>	>	>	>	>	>	>	<	<	<	<	>	<	
	-	>	>	<	<	<	>	>	>	>	>	>	>	>	<		<	<	>	<	
	*	>	>		>	<	>	>	>	>	>	>	>	>	<		<	<	>	<	
	/	>	>	>	>	<		>	>	>	>	>	>	>	<		<	<	>	<	
	(<	<	<	<	<	=	<	<	<	<	<	<		<	<	<	<	<	<	
)	>	>		>		>	>	>		>	>	>						>	<	
	<	<	<	<	<	<		>	>	>	>	>	>	>	<	<	<	<	>	<	
	>	<	<	<	<	<		>	>	>	>	>	>	>	<	<	<	<	>	<	
	<=	<	<	<	<	<		>	>	>	>	>	>	>	>	<	<	<	<	>	<
	>=	<	<	<	<	<		>	>	>	>	>	>	>	>	<	<	<	<	>	<
	==	<	<	<	<	<		>	>	>	>	>	>	>	>	<	<	<	<	<	<
	!=	<	<	<	<	<		>	>	>	>	>	>	>	>	<	<	<	<	<	<
	\$	<	<	<	<	<	0	<	<	<	<	<	<	<		<	<	<	<	<	<
	Num	>	>	>	>		>	>	>	>	>	>	>	>	>					>	
String	>	>	>	>	>	>		>	>	>	>	>	>						>		
null							>	>	>	>	>	>	>	>					>		
Identif	>	>	>	>		>	>	>	>	>	>	>	>	>					>		
is	<	<	<	<	<		<	<	<	<	<	<	<	<	<	<	<	<		<	
keyword	>	>	>	>	>	>		>	>	>	>	>	>	>					>		

obr. 3: precedenčná tabuľka

Sémantická analýza

[semantics.*]

Sémantickou analýzu jsme dělali za pomoci tabulky symbolů. Využili jsme několik tabulek symbolů, které jsme odkládali na stack vždy při vstupu do nového rozsahu platnosti. Při výstupu z rozsahu platnosti, tabulka byla ze stacku smazána.

Do tabulky jsme odkládali proměnné a funkce při jejich definici. Vždy před vložením jsme provedli kontrolu, že symbol již není v tabulce. Za předpokladu, že byl, program skončil sémantickou chybou. Při volání funkce, která ještě nebyla definovaná, byl vytvořen nový symbol s nastaveným boolem *declared* na false, a následně byl vložen do první tabulky symbolů společně s ostatními funkcemi. Za předpokladu, že je funkce definována později, je tento bool změněn.

Za předpokladu, že v programu narazíme na symbol, který je použit jako proměnná, do první tabulky je vložena na funkce s typem *T_GETTER* zase nastavený *declared* na false.

Po průchodu celého programu bereme první tabulku symbolů a provádíme několik sémantických kontrol. Tato tabulka musí obsahovat symbol pro funkci main(). Provedeme kontrolu volání funkce se špatným počtem parametrů, kde hledáme dva symboly - nedefinovanou funkci a další stejnojmenný symbol pro funkci. Procházíme tabulku a hledáme nedefinované symboly. Tyto chyby ukončí program sémantickou chybou.

Typová kontrola je v prekladači realizovaná v dvou fázích v závislosti od toho, či sú typy operandov známe v čase prekladu.

1. Statická kontrola: Prebieha v rámci precedenčnej analýzy. Funkcia *handleExpression* pri redukcii výrazov overuje vzájomnú kompatibilitu typov v prípade, že operandy sú literály.
 - Nepovolené operácie (napr. delenie reťazcov alebo aritmetika s null) sú detegované okamžite a vedú k ukončeniu prekladu s chybou 6.
 - Ak je operand premenná (*T_IDENTIFIER*), typová kontrola sa odkladá až na čas behu programu.

2. Dynamická kontrola: Túto kontrolu zabezpečuje generátor kódu. Pre operácie s premennými sú generované inštrukcie TYPE, ktoré dynamicky overujú typy hodnôt na dátovom zásobníku.
 - Výrazy: Pri binárnych operáciach generátor vkladá logiku pre polymorfizmus (napr. operátor + vykoná ADDS pre čísla alebo CONCAT pre reťazce). Neplatné typové kombinácie spôsobia behovú chybu 26.
 - Vstavané funkcie: generateBuiltinFunction generuje kód na kontrolu typov argumentov (napr. length vyžaduje typ string). Ak sa nezhodujú, nastáva chyba 25.

Tabulka symbolů

[symtable.*]

Symbol měl dva různé typy: proměnná nebo funkce. Funkce jsou dále odlišné podle enumu *function_type_t*, která sloužila k dělení na getter, setter a běžnou funkci. Jelikož ifj25 podporuje overloading, symboly v tabulce symbolů byly odlišné podle klíče *symbol_key*, který je tvořen pro funkce následovně: *foo_name* + “” + *param_num* + “_” + *foo_type*., kde *foo_name* je jméno funkce, *param_num* je počet parametrů a *foo_type* je typ funkce (g pro getter, s pro setter a f pro normální funkci). Proměnná byla uložena jenom pod svým jménem.

Tabulka symbolů byla implementována jako výškově vyvážený binární strom. Jednotlivé prvky mají hodnotu *height* a odkaz na levého a pravého potomka. Strom se sám vyvažuje. Vždy po vložení nového symbolu je provedena kontrola výšek podstromů. Za předpokladu, že strom je nevyvážený, tedy rozdíl výšek podstromů je větší než 1, musí se provést vyvažování, implementované ve funkci *balance_symtable()*, která se vždy volá po vložení nového symbolu *insert_symbol_to_symtable()*. Při vyvažování musíme prvně rozhodnout, která strana je delší a který potomek (levý/pravý) je nově přidaným prvkem, pak můžeme rozhodnout, jakým způsobem balancovat.

Máme dvě možnosti, buď rotovat doleva nebo doprava (*rotate_to_the_left()* a *rotate_to_the_right()*). Rotaci doleva (výměnu kořene pravým potomkem, přičemž původní kořen se stává levým potomkem nového kořene a levý potomek původního pravého potomka se stává pravým potomkem původního kořene, který je v tuhle chvíli již levý potomek) využijeme, když do pravé strany byl přiřazen nový prvek jako pravý potomek. Rotaci doprava (výměnu kořene levým potomkem, přičemž původní kořen se stává pravým potomkem nového kořene a pravý potomek původního levého potomka se stává levým potomkem původního kořene, který je v tuhle chvíli již pravý potomek), využijeme, když do levé strany byl přidán levý potomek.

Je zde ještě možné rotovat dvakrát. Rotaci doleva a pak doprava využíváme, když levá strana je delší a byl přidán pravý potomek. Prvně zrotujeme levou stranu doleva a pak celou část doprava. Rotaci doprava a pak doleva využíváme v opačném případě, tedy pravé část je delší a byl přidán prvek jako levý potomek. Pravou část kořene prvně zrotujeme doprava a pak celou část doleva.

Implementace logiky vyvažování stromu byla převzata z projektu z minulého roku týmu xkinzea00, kterou vytvořila xkrejce00.

AST

[ast.*]

Výstupom syntaktickej analýzy je Abstraktný syntaktický strom, ktorý je reprezentáciou samotného vstupného programu. V našom kompilátore je reprezentovaný ukazovateľmi na štruktúry typu `AstNode`, ktoré majú ďalej definované rôzne polia podľa ich daného typu `AstNodeType`. Pre túto štruktúru boli implementované viaceré pomocné funkcie, ako `printAst()`, `nodeTypeToString()` a `dataTypeToString()`.

Okrem týchto funkcií sú pre správnu funkčnosť AST implementované funkcie, ktoré manipulujú so samotnou štruktúrou stromu. Týmito funkciami sú:

- `initAst()`, ktorá inicializuje štruktúru AST vytvorením uzlu typu `NT_PROGRAM`,
- `createNode()`, ktorá vytvára uzol v AST a podľa požadovaného typu `AstNodeType` sú naplnené príslušné polia tejto štruktúry,
- `deleteNode()`, ktorá odstráni požadovaný uzol z AST a uvoľní pamäť, ktorú tento uzol zaberal,
- `deleteNodeList()`, ktorá rekurzívne volá funkciu `deleteNode()` a posúva sa na ďalší uzol, až kým nie je požadovaný podstrom prázdny.

V rámci syntaktického analyzátora je opakovane volaná funkcia `createNode()`, konkrétne sa tak deje pri rozpoznaní konštrukcie, pri ktorej nie sú vyvolané žiadne syntaktické chyby. Po vytvorení AST je táto štruktúra použitá v generátore kódu `IFJcode25`, ktorý z nej získava informácie o typoch uzlov a o konkrétnych hodnotách týchto uzlov (ak nejakú má, napr. pri literáloch).

Generátor

[generator.*]

Generátor kódu je poslednou fázou prekladu, ktorá transformuje abstraktný syntaktický strom (AST) na cieľový jazyk `IFJcode25`. Vstupom generátora je korektný AST a globálna tabuľka symbolov, výstupom je textový kód `IFJcode25` vypísaný na štandardný výstup.

Hlavnou vstupnou funkciou je `generateCode()`, ktorá najprv vygeneruje hlavičku `.IFJcode25`, potom definuje pomocné globálne premenné potrebné pre built-in funkcie a dočasné výpočty a následne rekurzívne prechádza AST. Pre jednotlivé typy uzlov (funkcie, výrazy, príkazy) existujú samostatné funkcie ako `generateFunction()`, `generateStatement()` a `generateExpression()`.

Generovanie výrazov je zásobníkové - výsledok každého výrazu sa ukladá na dátový zásobník. Pri binárnych operáciách sa vygenerujú oba operandy a vykoná sa operácia (napr. `ADDS`, `MULS`). Operátor `+` podporuje numerické sčítanie aj konkaténáciu reťazcov, pričom typ operácie sa určuje za behu.

Pre unikátne návestia pri vetvení a cykloch sa používa globálny čítač `label_counter`. Built-in funkcie s prefixom `lfj_` sú spracované vo funkcii `generateBuiltinFunction()`, ktorá obsahuje runtime typové kontroly a v prípade chyby generuje `EXIT` s príslušným kódom.

