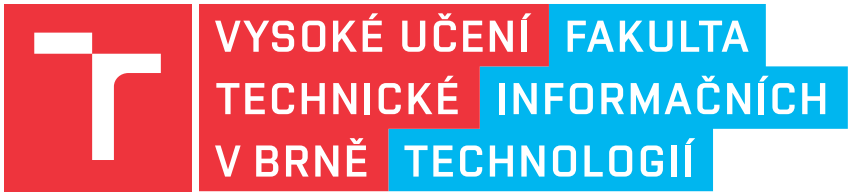


Projektová dokumentace
Implementace překladače imperativního jazyka IFJ20

Tým 75, varianta 1

Implementovaná rozšíření: **BASE**

Michal Pyšík	(xpysik00)	25 %
Karel Jirgl	(xjirgl01)	25 %
Václav Klem	(xklemv00)	25 %
Thanh Quang Tran	(xtrant02)	25 %

Obsah

1 Spolupráce v týmu

1.1 Rozdělení práce v týmu

Rozdělení práce probíhalo v době plánování. Nakonec jsme si všichni v různých částech implementace pomáhali, aby nebyl problém spojit všechny části předkladače v jeden funkční celek. Shodli jsme se na rovnoměrném rozdělení bodů, jelikož všichni členové týmu spolu aktivně komunikovali, pomáhali si a nebyly žádné významné neshody.

Michal Pyšík (xpysik00)

Implementace lexikálního analyzátoru a ustanovení typů tokenů

Karel Jirgl (xjirgl01)

Gramatika, Syntaktická a sémantická analýza, implementace tabulky symbolů, LL tabulky, precedenční syntaktické analýzy

Václav Klem (xklemv00)

Generátor cílového kódu

Thanh Quang Tran (xtrant02)

Gramatika, LL tabulka, tabulka precedenční syntaktické analýzy

Společná práce

Dokumentace, obecná struktura cílového programu

1.2 Způsob práce v týmu

Komunikace probíhala především prostřednictvím Discord serveru vytvořeného přímo pro tento projekt. Konaly se pravidelné porady a všichni členové týmu byli většinou k dispozici, pokud se vyskytl nějaký problém nebo bylo potřeba se dohodnout na konkrétních detailech implementace. Díky úspěšné komunikaci a vzájemnému porozumnění kódu mezi kolegy nebyl problém spojit jednotlivé části tak, aby spolu byly vzájemně kompatibilní. Pro správu zdrojových souborů jsme používali verzovací systém Git, většina práce se odehrávala pouze v hlavní větvi, jelikož v konkrétním souboru prováděla změny většinou jen jedna osoba najednou.

2 Lexikální analýza

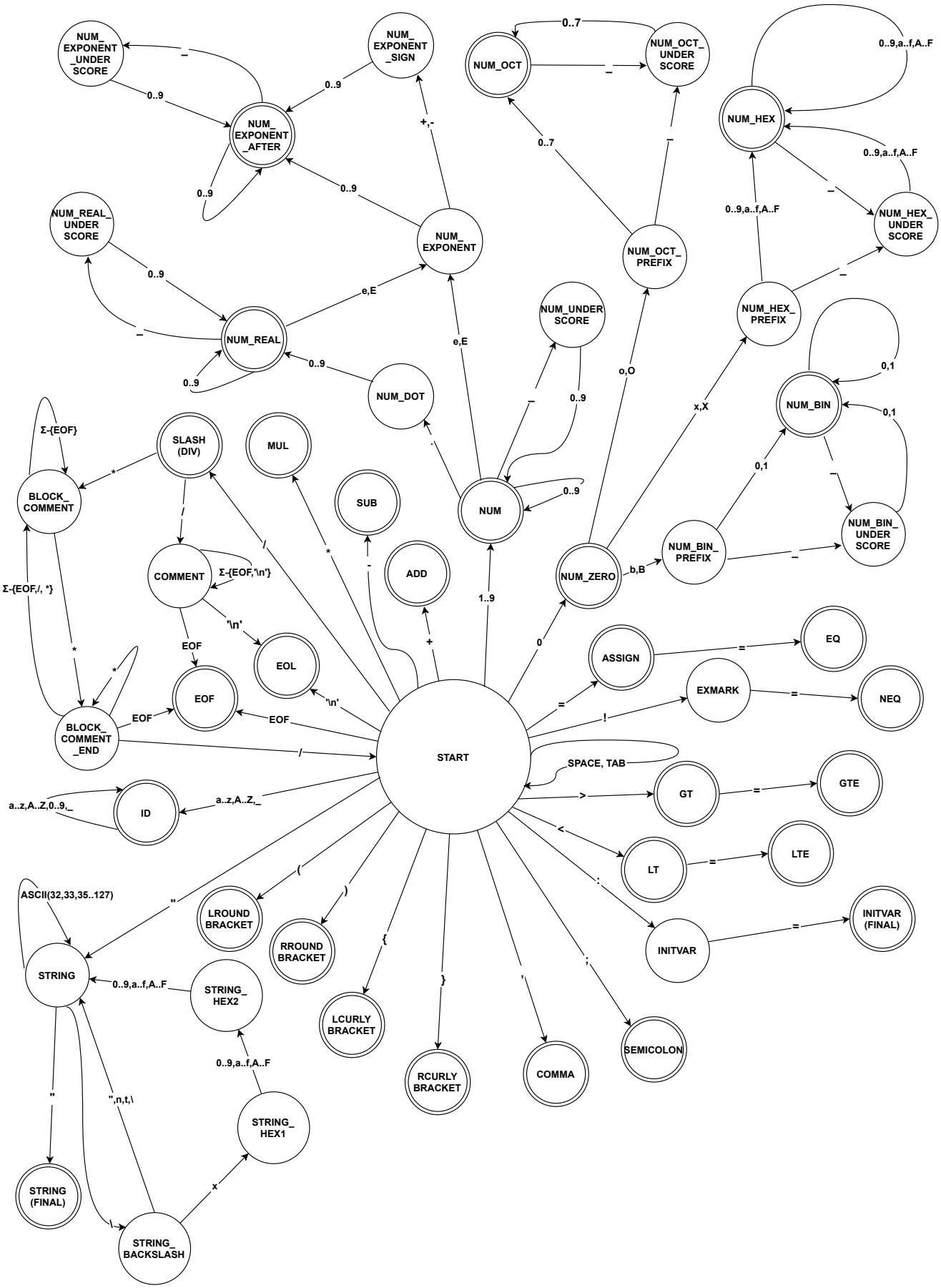
První částí překladače je lexikální analyzátor (scanner). Nejdůležitější funkcí je zde `scannerGetToken`, což je vlastně implementace deterministického konečného automatu, který čte jednotlivé znaky ze standartního vstupu, dokud nedojde do koncového stavu (návratová hodnota 0), nebo nenarazí na lexikální chybu (návratová hodnota 1). To je zajištěné pomocí přepínače `switch` umístěného v nekonečném `while` cyklu, kde každý `case` reprezentuje právě jeden stav automatu, a při každém opakování cyklu je čten právě jeden znak ze vstupu.

Funkce má jediný parametr, a to ukazatel na předem alokovaný prvek typu `Token`, jehož typ (a případně atribut) se nastaví podle přečtených hodnot a dosaženého koncového stavu. Mezi typy tokenů bez atributů patří například relační operátory, aritmetické operátory, přiřazení, klíčová slova, odřádkování (`EOL`), konec souboru (`EOF`), závorky a další povolené znaky.

Typy tokenů s atributy jsou pouze identifikátor, celé číslo, desetinné číslo a řetězec. Atribut tokenu je typu `union`, v případě identifikátoru nebo řetězce se tedy využívá pouze atribut `string`, u celého čísla `integer` a u desetinného čísla `real`. Když je zřejmé že načítáme nějaký z těchto 4 typů tak se přečtené znaky nejprve ukládají do bufferu (globální pole), a ve chvíli kdy je dosaženo koncového stavu, se do následující pozice v bufferu vloží nulový znak. Obsah bufferu až po tento znak je poté převeden na celé/desetinné číslo, nebo zkopírován do nově alokovaného stringu (jehož velikost pro alokaci je index pozice prvního nulového znaku v bufferu + 1). Tento postup je velice praktický, protože i když se obsah bufferu nemaže ale pouze přepisuje, cokoli za prvním nulovým znakem je ignorováno, například pozůstatky nějakého delšího dříve načteného řetězce. V případě ID se ještě zavolá funkce `keywordCheck`, která kontroluje zda se nejedná o klíčové slovo, v takovém případě uvolní alokovaný `string` a nastaví odpovídající typ tokenu.

Všechny načtené tokeny se ukládají do obousměrně vázaného lineárního seznamu `tokenList`, který je předáván dalším částem překladače. Z pracujících se seznamem stojí za zmínku `ScannerGetTokenList`, která plní seznam tokeny (voláním dříve popsané funkce), dokud nenarazí na token typu `EOF`, nebo volaná funkce nenahlásí lexikální chybu.

2.1 Graf konečného stavového automatu lexikálního analýzátoru



3 Syntaktická a sémantická analýza (parser)

Celá syntaktická a sémantická analýza se odehrává v souboru parser.c a konkrétně se spouští funkcí parserAnalyze.

V našem projektu je funkce parserAnalyze přímo volána z hlavní funkce main se vstupním parametrem tokenList, který obsahuje kompletní seznam tokenů načtený skenerem, který je implementován jako dynamický obousměrný lineární seznam. První akcí této funkce je vytvoření globální tabulky symbolů.

Proces parseru se skládá ze dvou hlavních funkcí:

- `parserPreRun` — zde dojde k naplnění globální tabulky symbolů uživatelskými a vestavěnými funkcemi, vytvoření jejich lokálních tabulek symbolů a uložení důležitých informací o nich (počet vstupních a výstupních parametrů a datové typy parametrů).
- `parserRunPredictiveSyntaxAnalysis` — spuštění samotného procesu syntaktické a sémantické analýzy.

3.1 Implementace funkce `parserPreRun`

Hned na začátku funkce dojde k naplnění globální tabulky symbolů vestavěnými funkcemi voláním funkce `parserSymTableInitBuiltIn`.

Dále se v cyklu prochází celý tokenList, kde se očekává token `TOKEN_KEYWORD_FUNC` následovaný tokenem `TOKEN_ID`, což značí začátek definice funkce. V globální tabulce symbolů se zkontroluje, že se nejedná o redefinici funkce a vloží se do ní záznam nové funkce, který bude obsahovat seznam typů vstupních parametrů a seznam typů výstupních parametrů funkce. Do záznamu se také přidá nová lokální tabulka symbolů, kam se přidají proměnné vstupních parametrů a proměnná blackhole (_).

Nakonec se kontroluje výskyt prologu (token `TOKEN_KEYWORD_PACKAGE`), výskyt hlavní funkce `main`, a že funkce `main` nemá žádné vstupní ani výstupní parametry.

3.2 Proces analýzy – funkce `parserRunPredictiveSyntaxAnalysis`

Na začátku funkce parserRunPredictiveSyntaxAnalysis dojde k nastavení ukazatele aktivního prvku `tokenList` na první prvek a spouští se analýza kódu.

3.2.1 Syntaktická analýza celého kódu

Pro syntaktickou analýzu kódu je použita Prediktivní syntaktická analýza. Na syntaktický zásobník je na počátku vložen neterminální stav `NONTERM_PROGRAM` a zároveň s `tokenList` je zásobník procházen v cyklu se třemi podmínkami:

1. Pokud je na zásobníku terminál `TERM_EOF`, zkontroluje se i aktuální čtený token, zda je roven tokenu `TOKEN_EOF`. V případě platnosti je analýza ukončena s pozitivním výsledkem — syntaxe i sémantika je správně. V případě, že na zásobníku je terminál `TERM_EOF`, ale aktuální token není `TOKEN_EOF`, je vrácena chyba špatného tokenu na vstupní pásce.

2. Jestliže se na zásobníku nachází neterminál, nalezne se podle aktuálního tokenu na vstupu v LL-tabulce číslo pravidla pro jeho rozklad. Pokud nebylo pravidlo nalezeno, došlo k syntaktické chybě. Jinak se ze zásobníku odstraní daný neterminál a nahradí se pravou stranou pravidla ze seznamu pravidel podle LL-tabulky. LL-tabulka i seznam pravidel je uložen v souboru common.h. Číslo pravidla je uloženo do zásobník s posloupností čísel pravidel pravého a levého rozboru.

3. Při terminálu na zásobníku jsou tři možnosti. Pokud terminálem je `TERM_PSEUDO_EPSILON`, pouze se zahodí ze zásobníku. Při neterminálu `TERM_EXPRESSION` se uloží do seznamu všechny tokeny výrazu a spustí se precedenční analýza výrazu. Jestliže terminál na zásobníku je roven tokenu ze vstupu, spustí se Sémantická analýza, zahodí se první neterminál na zásobníku a posune se ukazatel na vstupním seznamu tokenů na další token.

3.2.2 Syntaktická analýza výrazů

Syntaxe výrazů je kontrolována pomocí precedenční analýzy a precedenční tabulky (uložena v souboru common.h). Na začátku je na precedenční zásobník vložen pseudoterminál `TERM_PSEUDO_DOLLAR`, sloužící jako počáteční a ukončovací symbol. V cyklu `while` z precedenční tabulky získáme jednu ze tří operaci, která se má provést. Získáme ji z aktuálního tokenu na vstupu a nejvýše uloženého terminálu na precedenčním zásobníku. Pokud v precedenční tabulce není nalezena žádná operace, je vrácena syntaktická chyba. Před provedením některé ze tří operací se zkontroluje, že nejvýše uložený terminál na precedenčním zásobníku se nerovná `TERM_PSEUDO_DOLLAR` a aktuální načtený token se nerovná `TERM_PSEUDO_DOLLAR`. V opačném případě je voláno paralelně běžící Generování kódu výrazu a Precedenční analýza ukončena s kladným výsledkem.

1. Pokud je aktuální operace „=“ (rovná se), zavolá se paralelně běžící Sémantická analýza, na precedenční a sémantický zásobník se přidá aktuálně čtený token a nakonec se posune čtecí hlava tokenů na další token.

2. Při operaci „<“ (menší než) se provede to stejné, co při operaci „=“, ale k tomu navíc se přidá na precedenční zásobník terminál `TERM_PSEUDO_HANDLE`, který slouží pro označení začátku syntaktického pravidla pro další syntaktickou kontrolu.

3. Nastane-li operace „>“ (větší než), zavolá se na začátku paralelně běžící Generování kódu. Poté se najde v LL-tabulce pravidlo, které je rovno posloupnosti terminálů a neterminálů na precedenčním zásobníku od terminálu `TERM_PSEUDO_HANDLE` po vrchol zásobníku a nahradí se za levou stranu nalezeného pravidla. Pokud není pravidlo nalezeno, jedná se o syntaktickou chybu. Také je uloženo číslo pravidla do zásobník s posloupností čísel pravidel pravého a levého rozboru.

3.3 Sémantická analýza

Sémantická analýza běží paralelně se syntaktickou analýzou a její funkce parserSemanticAnalysis je volána pro každý přečtený token zvlášť. Sémantické chyby jsou kontrolovány pomocí podmínek, které očekávají určitý token nebo posloupnost tokenů anebo také stavy uložené v pomocných proměnných. Tyto podmínky můžeme rozdělit na globální a řádkové kontroly.

- **DEFINICE FUNKCE** – zkontroluje se, že je funkce definována v globální tabulce symbolů a její lokální tabulka symbolů se přidá na zásobník tabulek symbolů
- **VSTUP DO NOVÉHO ROZSAHU PLATNOSTI** – při levé složené závorce '{' nebo klíčovém slově for se zvýší úroveň zanoření a vytvoří se nová tabulka symbolů, která se vloží do zásobníku tabulek symbolů
- **UKONČENÍ ROZSAHU PLATNOSTI** – při pravé složené závorce '}' jednou nebo dvakrát při ukončení cyklu for se sníží úroveň zanoření a odstraní vrchní tabulka symbolů ze zásobníku tabulek symbolů
- **KONEC DEFINICE FUNKCE** - při pravé složené závorce '}' a pouze s poslední tabulkou symbolů (lokální tabulka symbolů funkce) se z kontroluje existence klíčového slova return
- **RETURN NENALEZEN** - při levé složené závorce '{' nastavíme returnExists na false
- **RETURN NALEZEN** - při klíčovém slově return nastavíme returnExists na true

3.4 LL tabulka

	package	id	EOL	func	(	)	EOF	,	{	}	return	expr	if	for	;	:=	=	int	float	string	INT	FLOAT	STRING	$
<program>	1		3	2			4																	
<param_in_first>		5				6																		6
<param_in_next>						8		7																8
<funkce_body>					10				9															
<param_out_next>						12		11																
<statements>		16	20							21	17		18	19										21
<statement_id>					22			25								23	24							
<id_next>								26									27							
<for_definition>		28													29									
<for_assignment>		30							31															
<for_assign_id>		33						32									33							
<id_express>		34										35												
<ids_express>		36										37												
<type>																		13	14	15				
<expr_next>			39					38	39															39
<arg_first>		43				44															40	41	42	44
<arg_next>						46		45																46
<arg>		50																			47	48	49	

3.5 Precedenční tabulka

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

3.6 LL – gramatika

1. <program> -> package id EOL <program>
2. <program> -> func id (<param_in_first>) <func_body> <program>
3. <program> -> EOL <program>
4. <program> -> EOF
5. <param_in_first> -> id <type> <param_in_next>
6. <param_in_first> -> ε
7. <param_in_next> -> , id <type> <param_in_next>
8. <param_in_next> -> ε
9. <func_body> -> { EOL <statements> }
10. <func_body> -> (<type> <param_out_next>) { EOL <statements> }
11. <param_out_next> -> , <type> <param_out_next>
12. <param_out_next> -> ε
13. <type> -> int
14. <type> -> float
15. <type> -> string
16. <statements> -> id <statement_id> EOL <statements>
17. <statements> -> return EXPRESSION <expr_next> EOL <statements>
18. <statements> -> if EXPRESSION { EOL <statements> }
    else { EOL <statements> } EOL <statements>
19. <statements> -> for <for_definition> ; EXPRESSION ; <for_assignment>
    { EOL <statements> } EOL <statements>
20. <statements> -> EOL <statements>
21. <statements> -> ε
22. <statement_id> -> (<arg_first>)
23. <statement_id> -> := <id_expression>
24. <statement_id> -> = <id_expression>
25. <statement_id> -> , id <id_next>
26. <id_next> -> , id <id_next>
27. <id_next> -> = <ids_expression>
28. <for_definition> -> id := <ids_expression>
29. <for_definition> -> ε
30. <for_assignment> -> id := <for_assign_id>
31. <for_assignment> -> ε
32. <for_assign_id> -> , id <for_assign_id>
33. <for_assign_id> -> = <ids_expression>
34. <id_expression> -> id (<arg_first>)
35. <id_expression> -> <expr>
36. <ids_expression> -> id (<arg_first>)
37. <ids_expression> -> <expr> <expr_next>
38. <expr_next> -> , <expr> <expr_next>
39. <expr_next> -> ε
40. <arg> -> INT
41. <arg_first> -> INT <arg_next>
42. <arg_first> -> FLOAT <arg_next>
43. <arg_first> -> STRING <arg_next>
44. <arg_first> -> id <arg_next>
45. <arg_first> -> ε
46. <arg_next> -> , <arg> <arg_next>
47. <arg_next> -> ε
48. <arg> -> FLOAT
49. <arg> -> STRING
50. <arg> -> id

4 Tabulka symbolů

Součástí překladače je tabulka symbolů. Do tabulky symbolů se ukládají informace o funkcích a proměnných, například jejich název, typ, návratové hodnoty a další. Je implementována jako binární vyhledávácí strom, kde každý uzel uchovává jméno funkce nebo proměnné. Pokud se jedná o funkci, tak se ukládá její počet a datový typ parametrů a návratových hodnot ukládané v obousměrně vázaném lineárním seznamu.

Binární vyhledávací strom `globalSymtable` ukládá informace funkcí a každý uzel ukládá svůj vlastní lokální tabulku symbolů `functionLocalSymTable`, která je taky implementována BVS a ukládá v nich proměnné a další informace obsažené v těle dané funkce.

5 Generace mezikódu IFJcode20

Generování cílového kódu IFJcode20 probíhá vždy po každém dokončení jedné iterace syntaktické a sémantické analýzy. Parser generátoru sekvenčně předává pravidla LL tabulky pravého a levého rozboru a sémantický stack, který obsahuje tokeny pro generování aritmetických a logických výrazů. Scanner poskytuje double-linked list tokenů z lexikálního analyzátoru.

Hlavní funkcí je `generatorGenerateCode`, která tiskne instrukce prostřednictvím přepínače `switch`, obstarávající generování podle tokenů. Proměnná `grammarRule` uchovává současné pravidlo, předané parserem a podmíněný cyklus `while`, jenž za pomoci sémantického zásobníku výrazů generuje instrukce aritmetických a logických operací.

Generování podmíněných příkazů `if` / `else` a cyklů `for` je kvůli správnému výpisu návěští `LABEL` implementováno zásobníkem celočíselných hodnot, které reprezentují jednotlivé ID daných klíčových slov. Jednotlivé buňky zásobníku rovněž obsahují inormace o současném kontextu v daném klíčovém slově (např. jestli se nacházíme ve větvi `if` nebo `else`). Ke správném tisku návěští dále napomáhají čítače ať už globální `globalBracketCnt` pro korektní tisk ukončení funkce `main`, nebo lokální `bracketCount` uložené v buňkách zásobníku ID logických bloků pro orientaci v hloubce zanoření.

K uchování informací o momentálním kontextu v kódu IFJ20 slouží `static` proměnné typu boolean, které jsou nastavovány pomocí již zmíněného přepínače `switch` a podmíněných výrazů, které se rozhodují na základě současného gramatického pravidla. Na práci s proměnnými jsou implementovány 2 zásobníky, `variableStack` pro práci s levou stranou přiřazovacího výrazu a `argumentStack`, pro správné předávání argumentů při volání funkcí.

6 Hlavní funkce main

Hlavní funkce překladače nejprve inicializuje `ErrorHandle`, což je pomocná datová struktura, jejíž úkolem je zpracovávat chybová hlášení. Dále se zde inicializuje obousměrně vázaný lineární seznam a načtou se do něj všechny tokeny. Na závěr se volá funkce `parserAnalyze`, která kromě syntaktické a sémantické kontroly volá také hlavní funkci generátoru mezikódu. Návratová hodnota celého programu se vyhodnotí voláním funkce `getResult`, která zpracovává již zmíněný `ErrorHandle`, který byl předáván všem klíčovým funkcím v rámce celého překladu.