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 týmu:

Adam Kinzel xkinzea00
Eliška Krejčíková xkrejce00
Natália Holbíková xholbin00
Juraj Mesík **xmesikj00**

rozšírenia: FUNEXP, BOOLTHEN

Obsah

Rozdělení práce

xmesikj00:
precedenčná analýza - psa.*; expr_stack.* ; precedence_table.*
lexikálna analýza - scanner.* ; error.*
syntaktická analyza - ast.*

xkinzea00:
syntaktická analýza - parser.* ; ast.*

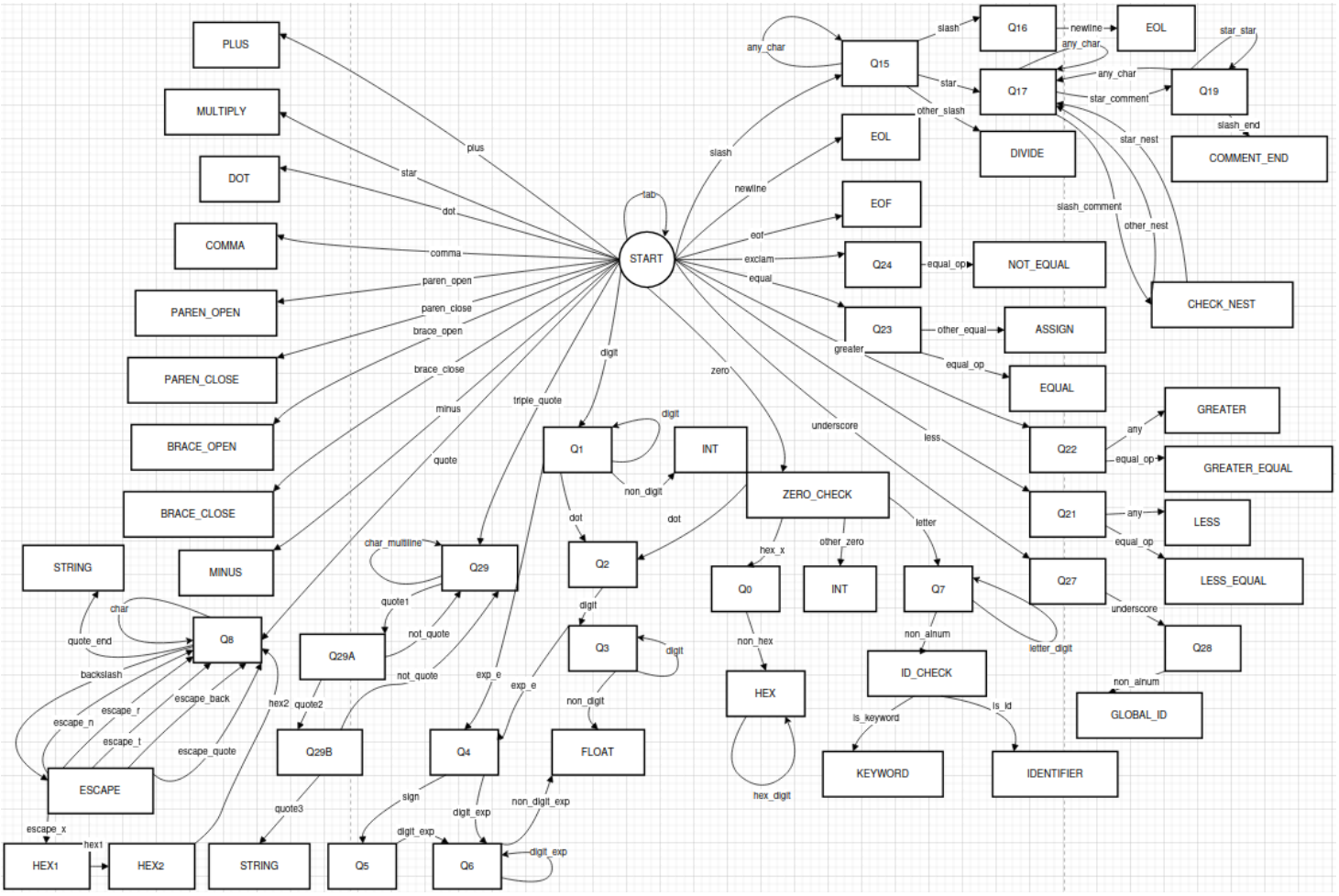
xkrejce00:
syntaktická analýza - parser.*
sémantická analýza - semantics.*
tabulka symbolů - symtable.*

xholbin00:
precedenčná analýza - psa.* (rozšírenie FUNEXP); expr_stack.*
lexikálna 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 jednoprůchodový rekurzivní sestup. Jedná se o skupinu funkcí, která odpovídá jednotlivým pravidlům gramatiky.

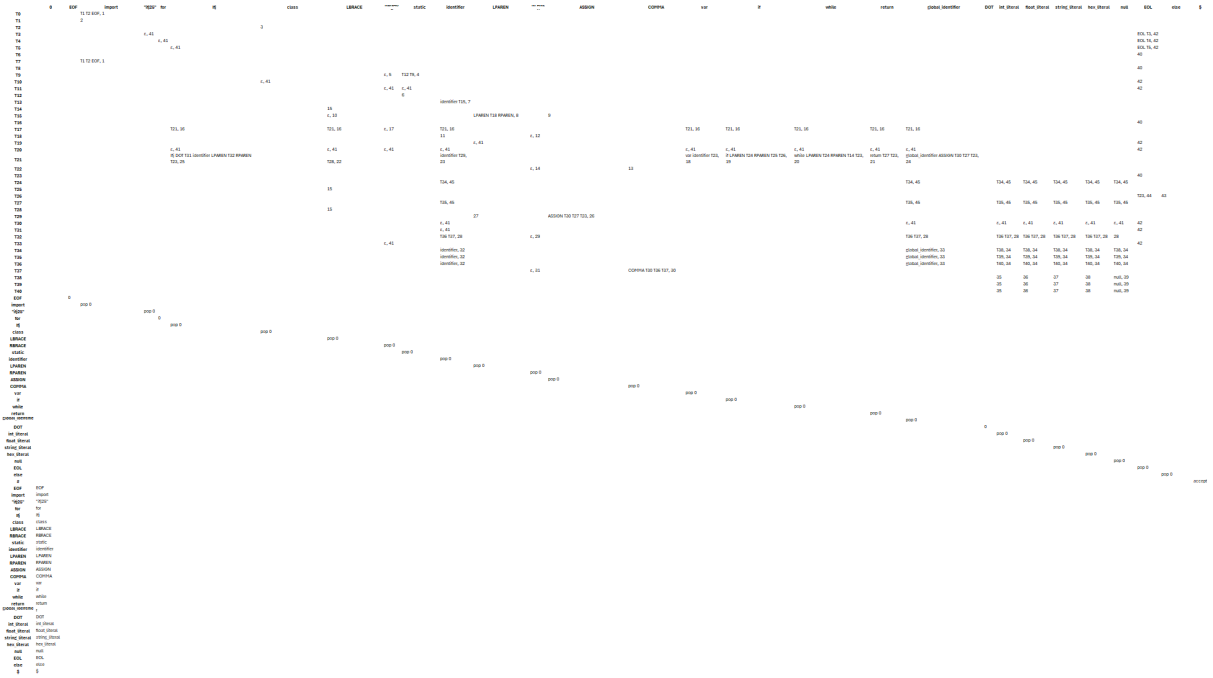
Postupně se jednotlivé funkce volají a využíváme funkci *getTokenSyntax()*, která získává token za pomoci funkce *getToken()* a zároveň filtruje komentáře. U jednotlivých tokenů jsme následně kontrolovali typ, často za pomoci funkcí *check_token()* a *check_token_by_str_value()*. Jestli některý token 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 → Prolog ClassDefinition EOF
(2) Prolog → import EolRuleNext "ifj25" EolRuleNext for EolRuleNext Ifj EolRule
(3) ClassDefinition → class Program { EolRule FunctionList }
(4) FunctionList → FunctionDef FunctionList
(5) FunctionList → ε
(6) FunctionDef → static FunctionHeader Block EolRule
(7) FunctionHeader → identifier FunctionHeaderTail
(8) FunctionHeaderTail → (ParameterList)
(9) FunctionHeaderTail → = EolRuleNext (identifier)
(10) FunctionHeaderTail → ε
(11) ParameterList → identifier ParameterListTail
(12) ParameterList → ε
(13) ParameterListTail → , EolRuleNext identifier ParameterListTail
(14) ParameterListTail → ε
(15) Block → { EolRule StatementList }
(16) StatementList → Statement StatementList
(17) StatementList → ε
(18) Statement → var identifier EolRule
(19) Statement → if (Expression) Block IfElse
(20) Statement → while (Expression) Block EolRule
(21) Statement → return Expression EolRule
(22) Statement → Block
(23) Statement → identifier StatementTail
(24) Statement → global_identifier = EolRuleNext Expression EolRule
(25) Statement → Ifj . EolRuleNext identifier (ArgumentList) EolRule
(26) StatementTail → = EolRuleNext Expression EolRule
(27) StatementTail → (ArgumentList) EolRule
(28) ArgumentList → Term ArgumentListTail
(29) ArgumentList → ε
(30) ArgumentListTail → , EolRuleNext Term ArgumentListTail
(31) ArgumentListTail → ε
(32) Term → identifier
(33) Term → global_identifier
(34) Term → Literal
(35) Literal → int_literal
(36) Literal → float_literal
(37) Literal → string_literal
(38) Literal → hex_literal
(39) Literal → null
(40) EolRule → EOL EolRuleNext
(41) EolRuleNext → EOL EolRuleNext
(42) EolRuleNext → ε
(43) IfElse → else Block EolRule
(44) IfElse → EolRule
(45) Expression → expression

LL tabulka:



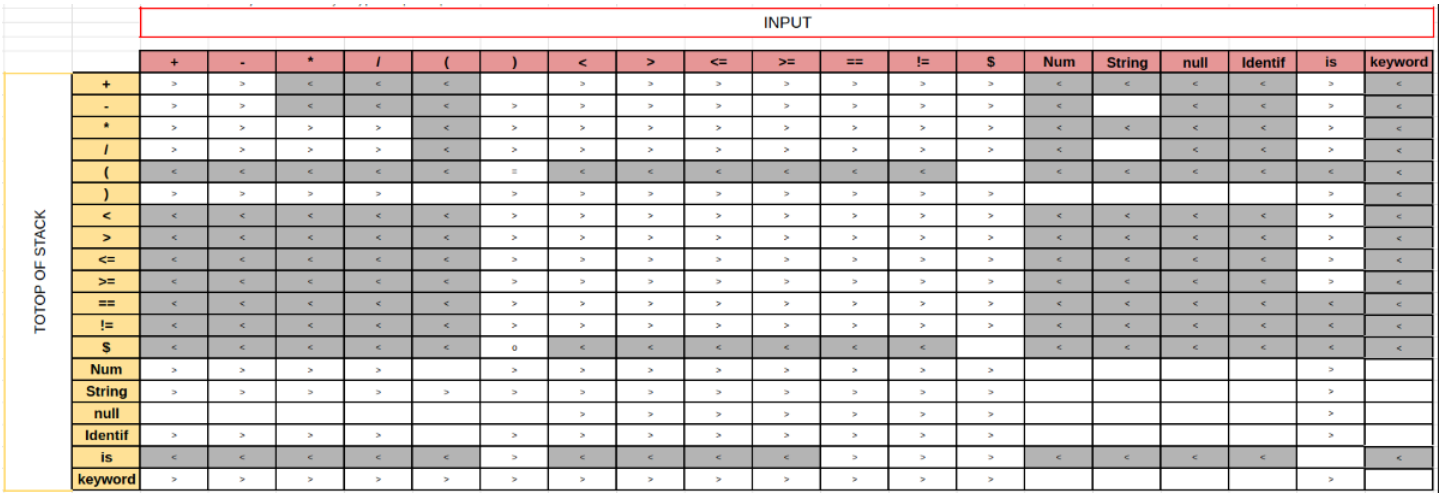
obr. 2: LL tabulka

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 funkcionalita na implementovaná 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 otváracej zátvorke. V pôvodnej verzii by sa správne vyhodnotil výraz, ale na zásobníku by ostal jediný terminál zatváracej zátvorky, čo by viedlo k chybovej hláške pri správnom vyraze. Museli sme teda nájsť spôsob ako odlíšiť, 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 vrátime do hlavného programu predošlý token, ktorý sme si ukladali práve kvôli tomuto prípadu.



TOTOP OF STACK \ INPUT	+	-	*	/	(	)	<	>	<=	>=	==	!=	$	Num	String	null	Identif	is	keyword
+	>	>	<	<	<		>	>	>	>	>	>	>	<	<	<	<	>	<
-	>	>	<	<	<	>	>	>	>	>	>	>	>	<		<	<	>	<
*	>	>	>	>	<	>	>	>	>	>	>	>	>	<	<	<	<	>	<
/	>	>	>	>	<	>	>	>	>	>	>	>	>	<		<	<	>	<
(	<	<	<	<	<	=	<	<	<	<	<	<		<	<	<	<	<	<
)	>	>	>	>		>	>	>	>	>	>	>	>					>	<
<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	<	<	<	>	<
>	<	<	<	<	<	>	>	>	>	>	>	>	>	<	<	<	<	>	<
<=	<	<	<	<	<	>	>	>	>	>	>	>	>	<	<	<	<	>	<
>=	<	<	<	<	<	>	>	>	>	>	>	>	>	<	<	<	<	>	<
==	<	<	<	<	<	>	>	>	>	>	>	>	>	<	<	<	<	<	<
!=	<	<	<	<	<	>	>	>	>	>	>	>	>	<	<	<	<	<	<
$	<	<	<	<	<	o	<	<	<	<	<	<		<	<	<	<	<	<
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 ukonční program sémantickou chybou.

Typová kontrola je v prekladači realizovaná v dvoch fázach 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 uložená 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 prípadně, 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 operáciach sa vygenerujú oba operandy a vykoná sa operácia (napr. ADDS, MULS). Operátor + podporuje numerické sčítanie aj konkatenáciu 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 Ifj. sú spracované vo funkcii generateBuiltinFunction(), ktorá obsahuje runtime typové kontroly a v prípade chyby generuje EXIT s príslušným kódom.