
# Projektová dokumentácia

## Implementácia prekladača imperatívneho jazyka IFJ20

Tým 035, variant I.

## Autori:


| | | |
|---|---|---|
| **Jakub Čuhanič** | xcuhan00 | 25% |
| Jakub Sokolík | xsokol14 | 25% |
| Tomaš Hak | xhakto01 | 25% |
| Kľučiar Adam | xkluci01 | 25% |


**Implementované rozšírenia:**
BOOLTHEN

9.12.2020

# Obsah

# 1 Úvod

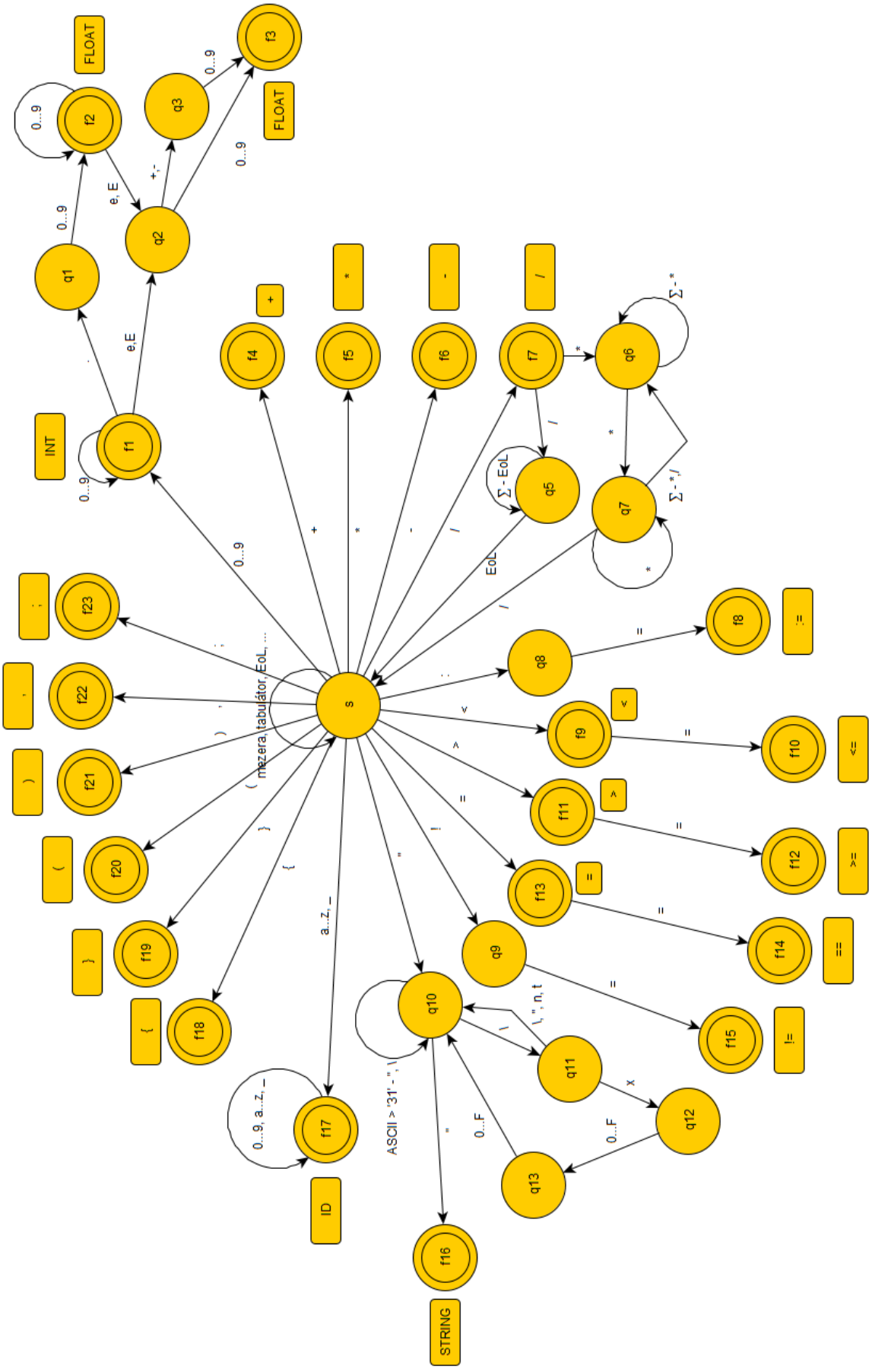
Cieľom dokumentácie je objasniť postup pri vypracovaní skupinového projektu z predmetov IFJ a IAL. Výsledný vypracovaný projekt pracuje so zdrojovým kódom v zdrojovom jazyku IFJ20, ktorý načíta zo štandardného vstupu a preloží ho do cieľového jazyka IFJcode20, teda generuje výsledný medzikód na štandardný výstup. V tejto dokumentácii nájdete postup pri riešení tohto projektu a popis implementácie jeho základných častí.

# 2 Návrh a implementácia

## 2.1 Lexikálna analýza

Lexikálna analýza je zodpovedná za načítanie tokenu. Jedná sa o konečný automat, ktorý je implementovaný vo funkcii *getToken*. Tá postupne načíta jednotlivé znaky zo vstupu, až do tej doby, pokiaľ automat nenarazí na reťazec, ktorý reprezentuje identifikátor, reťazcový, desatinný alebo celočíselný literál, operátor, podporovaný špeciálny znak alebo koniec súboru. Tieto údaje sú uložené v štruktúre *tToken*, ktorá obsahuje typ tokenu, reťazec obsahujúci načítané znaky, ktoré patria k tokenu, a eolFlag, ktorého hodnota označuje, či sa pred daným tokenom vyskytol koniec riadku. Štruktúra a aj typy tokenov sú definované v súbore *project.h*. Celá lexikálna analýza je implementovaná v súbore *lex.c*.

### 2.1.1 Konečný automat



## 2.2 Syntakticka analýza

Pri riešení projektu sme použili rekurzívny zostup. Nakoľko nám tento spôsob prišiel jednoduchší na implementáciu a porozumenie, rozhodli sme sa ho použiť, aj napriek tomu, že tento spôsob obsahuje väčšie množstvo numerických operácii, preto má väčšiu časovú náročnosť.

Syntaktická analýza úzko spolupracuje so sémantickou, precedenčnou a lexikálnou analýzou. Za týmto účelom bolo vytvorených niekoľko štruktúr. V súbore *project.h* je vytvorená štruktúra *tToken*, ktorá slúži na uchovanie typu, reťazca načítaný znakov a eolFlagu, ktorého hodnota označuje, či sa pred daným token vyskytol koniec riadku. Táto štruktúra sa využíva predovšetkým na komunikáciu s lexikálnou analýzou. Neskôr sme štruktúru rozšírili o položku *tToken savedToken*, keďže sa jedná o rekurzívnu štruktúru, doplnili sme štruktúru o sadu funkcií, ktoré korektne pristupujú k štruktúre a eliminujú výskyt chýb spôsobených nesprávnym zaobchádzaním. Túto položku využíva syntaktická analýza v špecifických prípadoch, kedy je potrebné načítať ďalšie tokeny a uchovať niekde v pamäti kľúčový token. Načítanie ďalšieho tokenu si syntaktická analýza vyžiada pomocou makra *GET_TOKEN*. Kontrola odriadkovania prebieha priamo v syntaktickej analýze, ktorá na základe syntaktických pravidiel presne vie, kde je odriadkovanie zakázané alebo potrebné. Následne sa za použitia makier *EOL_FORBID* a *EOL_REQUIRE*, kontroluje nastavený eolFlag.

Za účelom komunikácie so sémantickou analýzou bola vytvorená štruktúra *SymTable*, ktorá je popísaná v súbore *symtable.h*. Tá uchováva všetky potrebne informácie o premenných, funkciách a rámcoch. K štruktúre sme vytvorili celú sadu funkcií, pomocou ktorých syntaktická analýza vyvoláva sémantické kontroly.

Precedenčná analýza spracováva výrazy, ktoré syntaktická analýza ukladá do štruktúry *tokenList*. Výraz je reprezentovaný sériou tokenov, ktoré syntaktická analýza postupne pridáva do štruktury *tokenList*, zatiaľ čo striktne postupuje podľa gramatický pravidiel. Štruktúra je uložená v súbore *expr.h* a je k nej vytvorená sada funkcií.

Samotná syntaktická analýza je implementovaná v súbore *syntax.c* prostredníctvom funkcií, reprezentujúcich jednotlivé pravidlá. Vyvolanie syntaktickej analýzy prebieha v kmeňovom súbore *complier.c*, použité je na to volanie funkcie *rule_prog(tToken *token, tSymTablePtr STab, tKWPtr keyWords, bool* success)*. Štruktúra *keyWord*, je potrebná pre lexikálnu analýzu a teda ukazateľ na ňu sa v syntaktickej analýze nepoužíva, iba sa prostredníctvom syntaktickej analýzy predáva lexikálnej analýze.

Syntaktická analýza sa dokáže zotaviť z chýb. Medzi funkciami sa predáva ukazateľ na boolovskú premennú **success*. V prípade, že dôjde k syntaktickej chybe je tento flag nastavený na hodnotu false. Hodnota false znamená, ze funkcie prestanú vykonávať syntaktickú analýzu. V niektorých funkciách sa vyskytuje makro *CHECK_POINT*, ktoré v prípade, že daný flag je nastavený na false, žiadajú načítanie nových tokenov, až kým nenačítajú takzvaný zotavujúci token. Načítanie tohto tokenu, znamená že flag sa nastaví na hodnotu true a syntaktická analýza môže pokračovať ďalej a nebude ovplyvnená chybou, ktorá sa vyskytla. V prípade načítania tokenu označujúceho koniec súboru (EOF), flag naďalej ostáva nastavený na false, program korektne dobehne do konca, pozatvára všetky rekurzívne volané funkcie.

Nastavenie erroru v celom projekte je zabezpečené volaním funkcie *setError(int Error)* s parametrom, odpovedajúcim chybe, ktorá nastala. V syntaktickej analýze zodpovedá za nastavenie erroru už vyššie spomínané makro *CHECK_POINT*.

### 2.2.1 LL gramatika

| | | | |
|---|---|---|---|
| 1. | <prog> | -> | package main \n <func_def> \n <func_n> EOF |
| 2. | <func_def > | -> | func id (<params>) <return_types> { \n <body> \n } |
| 3. | <func_n> | -> | <func_def> \n <func_n> |
| 4. | <func_n> | -> | $\epsilon$ |
| 5. | <params> | -> | id <type> <param_n> |
| 6. | <params> | -> | $\epsilon$ |
| 7. | <params_n> | -> | , id <type> <param_n> |
| 8. | <params_n> | -> | $\epsilon$ |
| 9. | <return_types> | -> | (<type_r> <type_n>) |
| 10. | <return_types> | -> | $\epsilon$ |
| 11. | <type> | -> | int |
| 12. | <type> | -> | float64 |
| 13. | <type> | -> | string |
| 14. | <type_r> | -> | <type> |
| 15. | <type_r> | -> | $\epsilon$ |
| 16. | <type_n> | -> | ,<type_r> <type_n> |
| 17. | <type_n> | -> | $\epsilon$ |
| 18. | <body> | -> | <state> \n <body> |
| 19. | <body> | -> | $\epsilon$ |
| 20. | <state> | -> | <if> |
| 21. | <state> | -> | <for> |
| 22. | <state> | -> | id <state_n> |
| 23. | <state> | -> | _ <state_n> |
| 24. | <state> | -> | return <expr_opt> |
| 25. | <state> | -> | $\epsilon$ |
| 26. | <state_n> | -> | <func_call> |
| 27. | <state_n> | -> | <var_def> |
| 28. | <state_n> | -> | <var_asg> |
| 29. | <if> | -> | if <expr_bool> { \n <body> \n } <else> |
| 30. | <for> | -> | for <var_def_cycle>; <expr_bool>; <asg_opt> { \n <body> \n } |
| 31. | <else> | -> | else { \n <body> \n } |
| 32. | <else> | -> | $\epsilon$ |
| 33. | <func_call> | -> | (<param_actual>) |
| 34. | <param_actual> | -> | <term><term_n > |
| 35. | <param_actual> | -> | $\epsilon$ |
| 36. | <var_def> | -> | := <expr> |
| 37. | <var_def_cycle> | -> | id <var_def> |
| 38. | <var_def_cycle> | -> | $\epsilon$ |
| 39. | <var_asg> | -> | <id_n> = <values> |
| 40. | <values> | -> | <expr_wi> <expr_n> |
| 41. | <values> | -> | id <func_call> |
| 42. | <expr_n> | -> | , <expr><expr_n> |
| 43. | <expr_n> | -> | $\epsilon$ |
| 44. | <asg_opt> | -> | id = <expr> |
| 45. | <asg_opt> | -> | $\epsilon$ |
| 46. | <term> | -> | id |
| 47. | <term> | -> | INT_L |
| 48. | <term> | -> | FLOAT_L |
| 49. | <term> | -> | STRING_L |
| 50. | <term_n> | -> | ,<term><term_n> |
| 51. | <term_n> | -> | $\epsilon$ |
| 52. | <id_n> | -> | <valid_id><id_n> |
| 53. | <id_n> | -> | $\epsilon$ |
| 54. | <valid_n> | -> | id |
| 55. | <valid_n> | -> | _ |

*Pozn:*

| | |
|---|---|
| *<expr>* | *označuje výraz, s matematickými operátormi* |
| *<expr_bool>* | *označuje výraz, s matematickými operátormi a operátormi nerovnosti* |
| *<expr_wi>* | *označuje výraz, s matematickými operátormi, ktorý nezačína id* |

### 2.2.2 LL tabuľka

| | package | main | \n | EOF | func | id | ( | ) | { | } | , | ; | if | else | for | int | float64 | string | := | = | INT_L | FLOAT_L | STRING_L | _ | return |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| <prog> | 1 | | | | | | | | | | | | | | | | | | | | | | | | |
| <func_def> | | | | | 2 | | | | | | | | | | | | | | | | | | | | |
| <func_n> | | | | 4 | 3 | | | | | | | | | | | | | | | | | | | | |
| <params> | | | | | | 5 | | 6 | | | | | | | | | | | | | | | | | |
| <param_n> | | | | | | | | 8 | | | 7 | | | | | | | | | | | | | | |
| <return_types> | | | | | | | 10 | | 9 | | | | | | | | | | | | | | | | |
| <type> | | | | | | | | | | | | | | | | 11 | 12 | 13 | | | | | | | |
| <type_r> | | | | | | | | | | | 15 | | | | | 14 | 14 | 14 | | | | | | | |
| <type_n> | | | | | | | | 17 | | | 16 | | | | | | | | | | | | | | |
| <body> | | | 19 | | | 18 | | | | | | | 18 | | 18 | | | | | | | | | 18 | 18 |
| <stat> | | | 23 | | | 22 | | | | | | | 20 | | 21 | | | | | | | | | 24 | 25 |
| <stat_n> | | | | | | | 26 | | | | 28 | | | | | | | | 27 | | | | | | |
| <if> | | | | | | | | | | | | | 29 | | | | | | | | | | | | |
| <else> | | | 31 | | | | | | | | | | | 30 | | | | | | | | | | | |
| <for> | | | | | | | | | | | | | | | 32 | | | | | | | | | | |
| <func_call> | | | | | | | 33 | | | | | | | | | | | | | | | | | | |
| <params_actual> | | | | | | 34 | | 35 | | | | | | | | | | | | | 34 | 34 | 34 | | |
| <var_def> | | | | | | | | | | | | | | | | | | | 36 | | | | | | |
| <var_def_cycle> | | | | | | 37 | | | | | | 38 | | | | | | | | | | | | | |
| <var_asg> | | | | | | | | | | | 39 | | | | | | | | | | | | | | |
| <values> | | | | | | 41 | 40 | | | | | | | | | | | | | | 40 | 40 | 40 | | |
| <expr_n> | | | 43 | | | | | | | | 42 | | | | | | | | | | | | | | |
| <asg_opt> | | | | | | 44 | | | 45 | | | | | | | | | | | | | | | | |
| <term> | | | | | | 46 | | | | | | | | | | | | | | | 47 | 48 | 49 | | |
| <term_n> | | | | | | | | 51 | | | 50 | | | | | | | | | | | | | | |
| <id_n> | | | | | | | | | | | 52 | | | | | | | | | 53 | | | | | |
| <valid_id> | | | | | | 54 | | | | | | | | | | | | | | | | | | 55 | |

## 2.3 Sémantická analýza

Sémantická analýz je tvorená dvoma celkami, prvá časť sémantických kontrol prebieha počas syntaktickej analýzy a druha časť až po úspešnom ukončení syntaktickej analýzy. Pre komunikáciu so syntaktickou analýzou sme vytvorili štruktúru *SymTable*. Tabuľka slúži pre uchovanie všetkých informácii potrebných pre sémantickú analýzu a tiež pre správne generovanie kódu. Viac o tejto štruktúre sa môžete dočítať v kapitole 3.1 Tabuľka symbolov.

Zaujímavým riešením, ktoré sa vyskytlo v našom projekte je kontrola parametrov a návratových hodnôt funkcie. Jazyk IFJ20 je špecifický tým, že volanie funkcie nemusí striktne nasledovať za jej definíciou. Preto pri každom volaní funkcie alebo jej definícii sa parametre a návratové hodnoty súčasne vkladajú aj kontrolujú. Jazyk IFJ20 pozná totižto znak '_', ktorý znamená, že do neho priradená hodnota sa nikdy nepoužije. Preto napríklad volanie funkcie pred jej definíciou takýmto príkazom: *_, mod = div(a, 5)*, znamená, že funkcia vracia 2 hodnoty, ale poznáme iba typ jednej z nich. Preto je potrebné, aby sa tento jeden typ doplnil pri ďalšom volaní a zároveň sa skontrolujú už typy, ktoré poznáme.

V štruktúre *globalRec*, ktorá je súčasťou tabuľky symbolov, je niekoľko flagov, ktoré sú kontrolované po skončení syntaktickej analýzy a v prípade chyby s nastavuje chyba. Keďže volanie funkcie sa v jazyku IFJ20 môže vyskytovať pred jej definíciou, pridávanie parametrov a návratových hodnôt prebieha pri prvom volaní funkcie a následne pri jej definícii alebo ďalšom volaní prebieha kontrola.

## 2.4 Precedenčná analýza

Precedenčná analýza má na starosti spracovávanie výrazov. So syntaktickou analýzou komunikuje prostredníctvom štruktúry *SymTable*, z ktorej získava väčšinu potrebných inštrukcií. Potrebné tokeny, reprezentujúce výrazy uchováva syntaktická analýza do štruktúry *tokenList*. Následne, keď syntaktická analýza zistí, podľa pravidiel, koniec výrazu, volá precedenčnú analýzu funkciou *precedence*, ktorá token postupne spracováva. Po skončení tejto precedenčnej analýzy sa v štruktúre *tokenList* nachádza jediný prvok, ktorý následne syntaktická analýza posiela sémantickej analýze. Počas spracovávania výrazov precedenčná analýza rovno generuje

kód. Štruktúra *tokenList* je definovaná v súbore *expr.h* spolu so sadou funkcií používaných na spracovanie výrazov. Samotná precedenčná analýza je implementovaná v súbore *precedence.c*.

### 2.4.1 Tabuľka precedenčnej analýzy

| | + | - | * | / | ( | ) | ID | < | > | <= | >= | == | != | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| + | > | > | < | < | < | > | < | > | > | > | > | > | > | > |
| - | > | > | < | < | < | > | < | > | > | > | > | > | > | > |
| * | > | > | > | > | < | > | < | > | > | > | > | > | > | > |
| / | > | > | > | > | < | > | < | > | > | > | > | > | > | > |
| ( | < | < | < | < | < | = | < | < | < | < | < | < | < | |
| ) | > | > | > | > | | > | | > | > | > | > | > | > | > |
| ID | > | > | > | > | | > | | > | > | > | > | > | > | > |
| < | < | < | < | < | < | > | < | > | > | > | > | > | > | > |
| > | < | < | < | < | < | > | < | > | > | > | > | > | > | > |
| <= | < | < | < | < | < | > | < | > | > | > | > | > | > | > |
| >= | < | < | < | < | < | > | < | > | > | > | > | > | > | > |
| == | < | < | < | < | < | > | < | > | > | > | > | > | > | > |
| != | < | < | < | < | < | > | < | > | > | > | > | > | > | > |
| $ | < | < | < | < | < | | < | < | < | < | < | < | < | |

## 2.5 Telo prekladača

Telo prekladača je implementované v súbore *compiler.c*. Tento súbor nie je moc dlhý, pretože sa stará iba o správne alokovanie miesta pre štruktúry. Následne vyvolá činnosť syntaktickej analýzy prostredníctvom volania funkcie *rule_program*. Syntaktická analýza sa ďalej stará o všetky potrebne veci. Po skončení syntaktickej analýzy prekladač uvoľní miesto a vráti nastavenú chybu. Nastavenie chyby prebieha volaním funkcie *setError*, ktorá je implementovaná v súbore *error.c*. Pre získanie chyby je použitá funkcia *getError*, ktorá vracia chybu, ktorá sa pri preklade objavila ako prvá.

## 2.6 Generovanie kódu

Generovanie kódu prebieha volaním sady makier, ktoré sú implementované v súbore *inst.h*. Makrá pracujú so, štruktúrou *symTab*, a teda vedia vygenerovať správne inštrukcie jazyka IFJcode20. Niektoré makrá sú volané syntaktickou analýzou a iné sú volané precedenčnou. Jedinečnosť jednotlivých návestí a premenných zaručuje spôsob ich vytvárania. Každý jedinečný názov obsahuje číslo rámca a premennej, ku ktorej sa vzťahuje.

# 3 Algoritmy a dátové štruktúry

Počas tvorenia projektu sme implementovali niekoľko štruktúr, ktoré sú predstavené v tejto kapitole.

## 3.1 Tabuľka symbolov

Štruktúra *SymTable*, je definovaná v súbore *symtable.h*. Tabuľka slúži pre uchovanie všetkých informácii potrebných pre sémantickú analýzu a správne generovanie kódu. Samotná tabuľka sa ďalej delí na ďalšie dve tabuľky, pre ktoré sú vytvorené ďalšie štruktúry *globalRec* a *localRec*, definované v súbore *symtable_private.h*, tieto tabuľky sú implementované ako binárny strom, čím sme splnili zadanie našej varianty projektu.

*globalRec* obsahuje záznamy o funkciách, vrátane zabudovaných funkcií jazyka IFJ20, meno, parametre a návratové hodnoty funkcie. Pre uloženie návratových hodnôt a parametrov boli vytvorené dve štruktúry *retList* a *tokenList*, ktoré reprezentujú lineárny zoznam. *globalRec* ďalej uchováva niekoľko flagov, ktoré označujú, či daná funkcia bola definovaná, či vyžaduje návratové hodnoty alebo či obsahuje príkaz return.

*localRec* obsahuje záznamy o premenných, štrúktúra je implementovaná ako zásobník jednotlivých, medzi sebou previazaných, lokálnych rámcov. Každý rámec ma svoj binárny strom, v ktorom sú uložené premenné na rovnakej úrovni zanorenia. Kvôli vyhľadávaniu v nadradených rámcoch, obsahuje každý rámec ukazateľ na rámec nad ním. Každý rámec tiež obsahuje jedinečné číslo, ktoré slúži pri generovaní kódu k vytvoreniu jedinečných názvov premenných a návestí.

Pre tieto tabuľky je vytvorená sada funkcií, ktoré s nimi korektne pracujú, vykonávajú sémantickú kontrolu a v niektorých prípadoch aj generáciu kódu. Tieto funkcie sú implementované v súboroch *symtable.c*

a *symtable_private.c*, pri ich implementácii sme využili znalosti z predmetu Algoritmy a skúsenosti získané pri vypracovávaní projektov z tohto predmetu.

*SymTable* ako štruktúra ponúka príjemneérozhranie na prácu z vyššie uvedenými tabuľkami. Okrem ukazateľov na vyššie uvedené štruktúry, obsahuje ukazatele na aktívnu funkciu, jej aktívny parameter, aktívnu návratovú hodnotu a aktívnu premennú. Všetky tieto ukazatele umožňujú vykonávanie sémantických kontrol počas behu syntaktickej analýzy a to bez toho, aby syntaktická analýza musela vytvárať zoznamy tokenov.

### 3.2 Zoznam tokenov

Počas riešenia projektu sme potrebovali uchovávať zoznam tokenov. Za týmto účelom sme vytvorili štruktúru *tokenList*. Štruktúra obsahuje ukazateľ na token a ukazateľ na ďalší prvok, niekoľko flagov a číslo rámca. Najčastejšie túto štruktúru využíva precedenčná analýza, ktorej syntaktická analýza posiela zoznam tokenov reprezentujúcich výraz. Využitie však našla aj pri práci s viac-násobným priradením.

## 4 Práca v tíme

### 4.1 Spôsob práce a komunikácia v tíme

S vypracovaním projektu sme začali pomerne skoro. Zo začiatku bolo potrebné si vytvoriť ucelený pohľad na problematiku, ktorej sme dostatočne nerozumeli. Vzhľadom na neľahkú situáciu s vývojom ochorenia COVID-19, sme sa stretávali pravidelne po prednáškach z predmetu IFJ, prostredníctvom konferenčných hovorov na discorde. Po pár týždňoch, kedy sme si vytvorili dostatočný obraz o problematike syntaxou riadeného prekladača, sme sa pustili do jeho implementácie. Na jednotlivých častiach projektu sme pracovali vo dvojiciach, čo uľahčilo komunikáciu. Naďalej sme sa všetci stretávali prostredníctvom pravidelných konferenčných hovoroch. Ich cieľom bolo informovať o vývoji nášho projektu a tiež vytýčiť si úlohy na najbližšie obdobie. Frekvencia konferenčných hovor začala narastať, najvyššiu hodnotu dosiahla pár dní pred pokusným odovzdaním, kedy sme sa komunikovali každý večer. Táto frekvencia nám potom vydržala až do odovzdania projektu.

### 4.2 Verzovací system

Pre správu súborov sme použili verzovací systém Git. Zdrojové kódy sme mali uložené na vzdialenom repozitári webovej služby GitHub. Toto riešenie nám umožnilo prístup k najnovšej verzii nášho projektu. Počas práce na projekte sme využili možnosť vytvoriť vedľajšiu vetvu projektu, čo sa nám vyplatilo v prípade, kedy sme pripravovali projekt na pokusné odovzdanie a súbežne pokračovala práca na projekte.

### 4.3 Rozdelenie práce

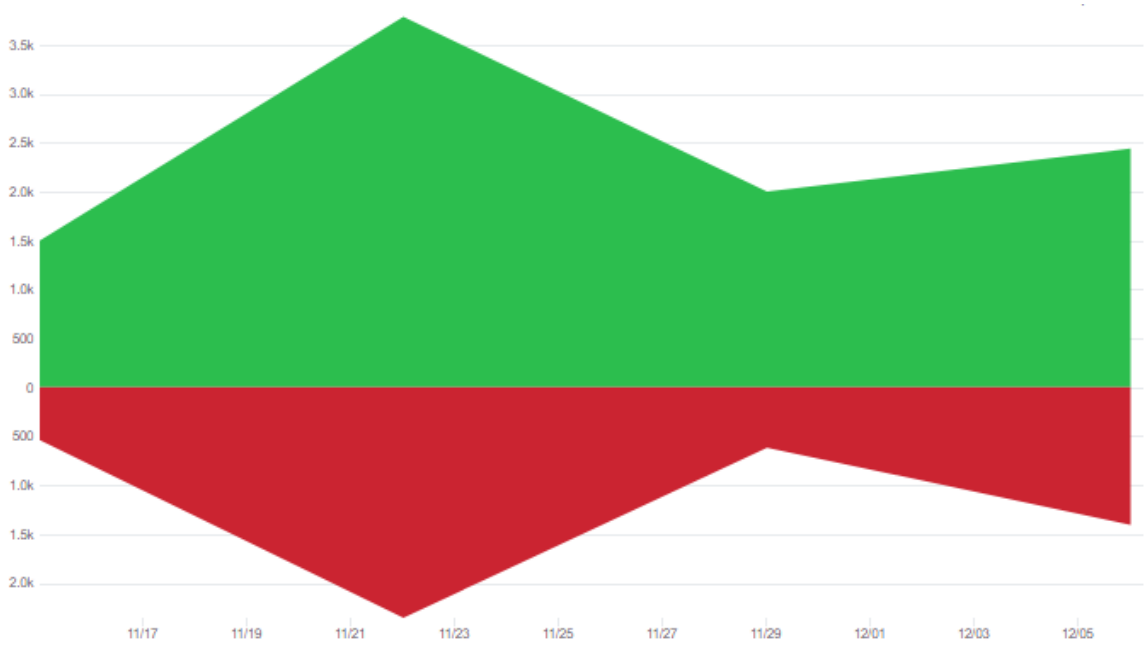
Na projekte sme pracovali vo dvojiciach. Dvojice sa striedali podľa toho, ktorú čas projektu bolo treba dokončiť. Každý z členov týmu vyvinul približne rovnaké úsilie, preto sme sa rozhodli body rozdeliť rovnomerne, každému členovi týmu 25%. V následujúcej tabuľke je uvedené kto sa podieľal na ktorej časti projektu.

| | |
|---|---|
| Jakub Čuhanič | vedúci tímu, komunikácia, tvorba LL gramatiky, tvorba LL tabuľky, Lexikálna analýza, generovanie kódu |
| Jakub Sokolík | tvorba a úprava LL gramatiky, syntaktická analýza, dokumentácia, generovanie kódu |
| Tomaš Hak | tvorba LL gramatiky, Lexikálna analýza, generovanie kódu, sémantická analýza, precedenčná analýza |
| Kľučiar Adam | syntaktická analýza, testovanie, debuging, tvorba prezentácie |

## 5 Štatistiky projektu

| | |
|---|---|
| počet riadkov | 4100 |
| počet commitov | 230 |
| deň s najväčším počtom commitov (35) | 8.12. |

počet pridaných a odstránených riadkov za týždeň:




# 6 Záver

Nakoľko sa jedná o náš prvý veľký projekt a tiež o prvú skúsenosť s prácou v tíme, zadanie projektu nás vystrašilo a nevedeli sme si predstaviť ako vypracujeme také množstvo požiadaviek. Postupom času a vďaka prebratej látke týkajúcej sa projektu, sme si problematiku vedeli predstaviť a navrhnúť riešenia daných celkov. Keďže sme projekt vypracovávali súbežne s preberajúcim učivom, niektoré riešenia sú zbytočne komplikované. Tento fakt sme však zistili až s odstupom času a implementácia nového, menej komplikovaného, riešenia by nás stála veľa drahocenného času. Preto vypracovanie projektu nie je dokonalé a dalo by sa zlepšiť po viacerých stránkach.

Spoločne sme však zhodnotili, že projekt bol zaujímavý a bol pre nás prínosný po viacerých stránkach. Najviac nás posunulo dopredu to, že sme pracovali v tíme. Komunikácia bola dôležitá nie len počas konferenčných hovorov, ale aj počas vzájomnej revízie kódu. Tento aspekt viedol k tomu, že sme si medzi sebou vymenili cenné skúsenosti a zlepšili tak svoje programovacie schopnosti. Počas práce na projekte sme sa tiež zdokonalili v používaní vzdialeného repozitára.

# 7 Zdroje a použitá literatúra

A Prednášky a demo cvičenia z predmetu Formálne jazyky a prekladače

B Demo cvičenia z predmetu Formálne jazyky a prekladače z predchádzajúcich rokov

C Prednášky z predmetu Algoritmy