1. SLAJD – Úvod
Vážená komise dobrý den,
dnes vám odprezentuji naše řešení projektu z předmětů Formální jazyky a překladače a Algoritmy. Na projektu jsem spolupracoval společně s *vyjmenuj ostatní kolegy – u Bárta zmiň, že je vedoucí týmu.*

2. SLAJD – Struktura překladače
Hlavní části překladače se nachází v céčkových souborech scanner, parser, expression a generate_code. V parser a expression jsme implementovali jak syntaktickou, tak sémantickou analýzu a v symtable je potom implementovaná tabulka symbolů. Překlad začíná voláním parseru, ze kterého se potom volají další funkce a komponenty.

3. SLAJD – Lexikální analyzátor
Lexikální analyzátor jsme implementovali konečným automatem s využitím přepínačů. Zdrojový soubor ze vstupu se čte znak po znaku a po úspěšném přečtení lexému **naplní scanner strukturu tokenu, kterou mu posílá parser**.

Hlavním atributem tokenu je jeho **identifikátor**, podle kterého se později určí co s ním. Následuje jeho hodnota **Value**. Je tvořena **unií** a může nabývat jednoho ze 4 datových typů: integer, double, dynamický string, nebo výčtového typu keyword.

Struktura **dynamického stringu** je řetězec, který se sám zvětší, když se naplní jeho alokovaná paměť. Jde tedy vlastně o řetězec proměnné délky. Pro toto řešení jsme se rozhodli kvůli předem neznámé délce vstupního řetězce.

Klíčovou částí bylo odladit správné ukončení čtení při neočekávaném konci souboru. Například konec souboru před ukončovacími uvozovkami řetězcového literálu.

V průběhu další implementace jsme objevili, že bylo zbytečné generovat token pro bílé znaky.

4. SLAJD – Syntaktická shora dolů
Pro syntaktickou analýzu jsme zvolili **rekurzivní sestup pro jeho jednoduchost**. To se později obrátilo proti nám, když jsme při implementování nacházeli v pravidlech chyby. Každá z funkcí představující neterminály si předává strukturu tokenu a vrací hodnotu odpovídající chybě při překladu, nebo úspěchu. V případě chyby se chybový kód funkcemi předává až do main souboru ifj2021.c, kde se ukončí.

Aby se mohl překladač rozhodnout, které z pravidel použít, zavolá **scanner, který mu naplní token**. Podle něj se rozhodne co dál a **čtení** vstupního souboru se **vrátí** před tento rozhodovací token.

Podle pravidel bylo předem jasné, ve kterou chvíli se bude vykonávat analýza **zdola nahoru** a v tu chvíli se zavolá **funkce exprSyntaxCheck**.

5. SLAJD – Syntaktická zdola nahoru

Syntaktickou analýzu zdola nahoru jsme dle zadání zvolili precedenčním přístupem. Pro její chod jsme vytvořili **zásobník**, do kterého se přidávají **terminály a neterminály**. Podle terminálu nejblíže vrcholu a znaku na vstupu se použije pravidlo z dříve vytvořené precedenční tabulky. Precedenční tabulku jsme implementovali dvourozměrným polem.

V případě, že pravidlo nakazuje redukci výrazu, zkontrolují se pravidla gramatiky a redukce se provede.

Jelikož jazyk ifj21 nemá žádný znak udávající konec příkazu, tak se **kontrola výrazu dokončí**, pokud na vstup dojde **neočekávaný token** a řízení se vrátí syntaktické analýze shora dolů.

6. SLAJD – Tabulka symbolů

Při implementaci tabulky symbolů jsme využili dvou datových struktur. Jednotlivé **bloky programu** ifj21 jsou reprezentovány **položkami v zásobníku**, který jsme realizovali lineárním seznamem. Zásobníkem se při vyhledávání dá **procházet sekvenčně** položkou po položce, díky tom nemusíme pracovat jen s prvkem na vrcholu. Každý prvek má potom ukazatel na vrchol binárního vyhledávacího stromu.

Binární vyhledávací strom nám slouží k ukládání identifikátorů funkcí a proměnných. Každý **uzel s identifikátorem obsahuje doplňující data**, např. zda byla proměnná inicializována, nebo u funkce lineární seznam s parametry a návratovými hodnotami a další.

Tabulka symbolů jako taková obsahuje **dva ukazatele**. Jeden vždy ukazuje jen na **globální rámec**, který ukazuje na strom s funkcemi. Druhý ukazuje na **nejvrchnější rámec** (blok) programu s proměnnými.

7. SLAJD – Sémantická analýza

Sémantické kontroly **probíhají souběžně se syntaktickou** analýzou. Ty probíhají při práci s proměnnými a funkcemi. Řešili jsme např. kontroly **redefinice** funkcí a proměnných, **datové typy** u přiřazení hodnoty a zda odpovídá počet parametrů u volání funkce.

Pro sémantickou kontrolu **výrazů** bylo třeba doimplementovat **zásobník datových typů**, aby bylo možné zjistit, zda jsou datové typy operandů kompatibilní a případně zajistit jejich konverzi.

8. SLAJD – Generování kódu
Kód **generujeme postupně voláním funkcí** z generate_code. Každá z funkcí obsahuje několik výpisů instrukcí na standardní výstup. Naší snahou bylo udělat je znovu použitelné a univerzální. Výstup **některých funkcí je statický** a nemění se (**např. kód vestavěných funkcí**) a některé funkce dostávají jako **parametr identifikátor** a lze je využít ve více případech (**např. volání funkce**).

Stínění jsme se rozhodli rozklíčovat, tak že do **dvojitě vázaného seznamu se uložily všechny identifikátory** proměnných, které se následně nadefinovaly **do nového dočasného rámce**. Případně se zkopírovaly i jejich hodnoty. Pokud se hodnota v dočasném rámci změnila, **novou hodnotu jsme distribuovali zpět** do nižšího rámce.

Vypsání vestavěných funkcí jsme dali do samostatného souboru builtIn.c

9. SLAJD – Práce v týmu
Řešení projektu jsme komunikovali nejprve aktivně na Discordu, kam jsme si ke konci psali jen připomínky k chybám a hlavním kanálem se stal messenger. Nejčastěji jsme se ale domlouvali osobně, nejen proto, že se už dlouhou dobu všichni známe, ale i proto, že bydlíme na stejném patře kolejí.

K týmové práci jsme aktivně využívali verzovací systém git. Tady se zúčtovaly znalosti nabyté v předmětu IVS a vyzkoušeli jsme si i **práci s pull requesty**. Od minulého semestru jsme také k **zpřehlednění git historie** použili ke každé zprávě **gitmoji**, které představuje nějakou logickou změnu. Pro probrání důležitějších témat jsme si rezervovali místnosti přímo na fakultě, ale výrazně nedoporučujeme zůstavat až do desáté hodiny, **zamyká se brzo.**

10. SLAJD - Závěr
Tímto bych Vám chtěl poděkovat za pozornost a nechávám prostor pro dotazy.