* Základní dělení na lexikální analyzátor, tabulku symbolů, syntaktický analyzátor, sémantický analyzátor a generátor.
* Tokeny se vytváří v scanneru, jejich syntaxe (pořadí) v parseru, další parametry v pedantovi (deklarováno/definováno), a končí při generování kódu v generátoru. Tabulka symbolů je zejména pro práci sémantické analýzy a generování
* Lexikální analyzátor bere ze vstupu znaky, rozděluje je na lexémy, které reprezentuje tokeny. Samotná struktura lexikálního analyzátoru odpovídá přibližně typům lexémů.
  + Token je záznam, skládající se z typu (konstanta, identifikátor, klíčové slovo, operátor, separátor, konec souboru) a hodnoty, kterou je buď identifikátor do tabulky, nebo adresa řetězce (v případě identifikátoru)
  + Pro alokaci řetězce používá scanner dynamický buffer.
  + Sám pro načítání užívá obálky nad vstupy a výstupy, která mu umožňuje vracet byty na vstup.
* Tabulka symbolů, definovaná v souboru *symtable.c*
  + Reprezentována hashovací tabulkou, jak je dáno v zadání, přičemž celkově je tabulka dána jako pole záznamů, reprezentujících funkce, které mimo data jako jméno, spojitý seznam parametrů, a návratový typ obsahuje také hashovací tabulku proměnných
  + Obě Tabulka má pouze jeden záznam na index, pokud se zaplní více, než z poloviny, automaticky se zvětšuje. Proto nelze přistupovat indexem, jelikož je proměnný, identifikátorem je řetězec.
* Syntaktický analyzátor byl implementován rekurzivním sestupem, kde na základě prvního tokenu se ve funkcích vyšší logiky volají funkce dále (jako například z funkce zpracování řádku se nalezne klíčové slovo *dim* a volá se funkce pro zpracování deklarace proměnné).
  + Pro svoje potřeby také využívá zásobníku tokenů ve scanneru.
  + Obsahuje také ExpressionParse, vyhodnocovač výrazů, kterému ve vhodných chvílích předává řízení
* Sémantický analyzátor je místo, kde ExpressionParse zanechá zásobníky s výrazy pro další typovou kontrolu a generování.
  + Do zásobníku se přidají na vhodná místa speciální přetypovávací tokeny. Při generování se berou postupně tokeny a generuje se kód, který výraz vyjádří na zásobníku cílového jazyka.
  + Sémantický analyzátor je také výhodný, protože část kódu odnáší z *parseru*, který je už tak docela velký.
* Generátor cílového kódu obsahuje dva základní zásobníky: zásobník stavů generátoru a zásobník návěští. Díky tomu jsou pouze dvě funkce pro přenášení dat: pro zásobníky tokenů, nebo samostatné tokeny.
  + Naše řešení generuje přímo výsledný kód, bez použití optimalizátoru.
* Následuje demonstrace koordinace parseru, pedanta a generátoru.
  + Na vstupu je daný kus kódu. Parser načte klíčové slovo a umístí do stavového zásobníku generátoru stav podmínky.
  + Následně ExpressionParse zpracuje výraz, odesílá postupně do Pedanta, kde se vkládají tokeny do zásobníku, kontroluje se, zda byly proměnné definovány, typová kompatibilita, popř. přetypování.
  + Na konci zde zůstává celkový typ zásobníku a zásobník, Parser jej odešle do generátoru a ten z něj vygeneruje kód.
  + Následně parser zpracuje porovnávací znaménko a znovu volá ExpressionParse.
  + Ten obdobně vytvoří zásobník a Pedant poté porovná typy, provede přetypování a vygeneruje se i druhý zásobník.
  + Parser poté odešle token s porovnávacím znaménkem do generátoru, který díky stavovému zásobníku pozná, že se jedná o znaménko porovnání a vygeneruje porovnání.
  + K tomu používá zásobník návěští, na který se uloží návěští, které je třeba vygenerovat pro skok za podmínku.
  + Dále se kontroluje syntax zbytku příkazu a těla podmínky.
  + Když se dosáhne konce podmínky, dá se vědět generátoru, ten na základě stavového zásobníku a zásobníku labelů generuje návěští pro skok.
  + Pro ilustraci v případě cyklu se generují návěští dvě, jedno před podmínkou, ze kterého se skáče na konci těla cyklu, druhé za cyklus, na které se skáče na základě podmínky.
* Ještě taková zajímavost. Pro správu alokovananých tokenů jsme použili modul, který si všechny alokované adresy pamatuje a uvolňuje.