Dobrý den, já jsem Pavel Hýža, toto jsou mí kolegové David Krejčí, Lukáš Farkašovský a vedoucí týmu Jan Kalina. Pro práci na projektu jsme se osobně scházeli jednou týdně ve studovnách, jinak jsme na komunikaci používali Discord, kde jsme měli organizované kanály pro jednotlivé problémy. Samotný projekt jsme vyvíjeli v prostředí Visual Studio Code a jako systém pro správu verzí jsme použili GitHub.

Náš vývoj začal **lexikálním analyzátorem**, který v našem projektu najdete pod jménem Scanner a pracoval jsem na něm hlavně já. Scanner nejprve třídí vstupní znaky do 7 námi definovaných skupin. Písmena, číslice, bílé znaky, jednoduché operátory, složité operátory, znak EOF a znaky mimo jazyk IFJ24. Díky těchto definicím je možné například využívat znaky mimo základní tabulku ASCII a další v komentářích nebo ve znakových řetězcích.

Pro uchování obsahu tokenů identifikátorů, stringů nebo číselných tokenů se používá knihovna pro **dynamické řetězce**, s níž lze vytvořit řetězec potřebné délky. Ten je navržen tak, aby se při překročení aktuální délky, automaticky zvětšil. Pracoval na něm hlavně Lukáš.

David mezitím pracoval na naší **tabulce symbolů**. V tabulce jsme pro hashovací funkci použili algoritmus djb2. Pokud nastanou indexové konflikty, řešíme je lineárním průchodem. Naše tabulka je implementována dynamicky a díky tomu je náš překladač schopen zpracovávat libovolně rozsáhlé programy s vysokou paměťovou efektivitou. Pro každý rámec v parsovaném kódu vytváříme jednu tabulku symbolů, ty uchováváme na **zásobníku rámců**, který je implementován jako lineárně vázaný seznam. Při vynoření je rámec odebrán ze zásobníků, ale zůstává v poli rámců pro využití během sémantické analýzy.

Pokračovali jsme vývojem **syntaktického analyzátoru**, který v našem projektu najdete pod jménem Parser. Skládá se z abstraktního syntaktického stromu a 5 submodulů, na kterých pracoval hlavně Honza. Jedná se o LL a precedenční tabulku, LL a precedenční parser a společnou knihovnu parser common.

Nejprve jsme začali návrhem **LL gramatiky** pro **LL tabulku**. Při návrhu jsme začali tvorbou neterminálů pro hlavní konstrukce jazyka, ty jsme následně dělili na menší, například kvůli odstranění levé rekurze. David vytvořil vlastní C++ program pro automatické generování některých množin, což nám velmi usnadnilo další práci na projektu. Při realizaci LL tabulky jsme uvažovali nad řešením pomocí dvourozměrného pole, ale rozhodli jsme se pro implementaci vlastní struktury. Ta obsahuje klíč a pole LL pravidel, ve kterém je vyhledáváno pomocí binárního vyhledávání.

Dále jsme vytvořili **abstraktní syntaktický strom**. Při návrhu jsme se rozhodli namísto jednoho univerzálního uzlu vytvořit minimální počet různých uzlů kvůli přehlednosti. Konstrukce stromu probíhá zdola nahoru během parsingu. Pro zajištění vysoké volnosti při konstrukci AST, obsahují struktury některých uzlů ukazatel na void, což nám umožnilo propojit tyto uzly s různými typy AST uzlů. Pro vizualizaci AST jsme vytvořili vlastní program.

Prvním hlavním submodulem parseru je **LL parser**. Provádí LL syntaktickou analýzu pomocí rekurzivního sestupu, řídí Scanner a pro zpracování výrazů propůjčuje řízení precedenčnímu parseru. Vedle kontroly syntaktické správnosti parsovaného kódu, plní v rámci parsingu tabulku symbolů, a dokonce provádí také základní sémantickou kontrolu jako je kontrola zastínění nebo redefinice identifikátoru v rámci jejich rozsahu platnosti.

Druhým hlavním submodulem parseru je **precedenční parser**, kterému je LL parserem propůjčováno řízení při parsování výrazů nebo argumentů funkcí v rámci rozšíření FUNEXP, které jsme se rozhodli implementovat. Pracuje s precedenční tabulkou priorit operátorů a redukčními pravidly tvořenými sekvencí symbolů. Součástí je také náš vlastní precedenční zásobník a také „zásobník precedenčních zásobníků“ využívaný při parsování argumentů volání funkcí v rámci rozšíření FUNEXP. Pokud při žádosti o provedení redukce na precedenčním zásobníku dojde ke shodě mezi sekvencí symbolů na vrcholu zásobníku a sekvencí některého z redukčních pravidel, aplikuje se dané redukční pravidlo, jinak dojde k syntaktické chybě. Naše precedenční tabulka obsahuje také priority symbolů tvořících volání funkcí; ty mohou vystupovat ve výrazech a argumentech volání funkcí v rámci rozšíření FUNEXP.

Obecné funkce využívané ve všech submodulech parseru jsou obsaženy v submodulů **parser common.** Jeho cílem je minimalizovat duplicitní kód mezi různými submoduly parseru. Poskytuje jednotné zázemí pro LL i precedenční parser. Součástí je speciální funkce obsahující statické proměnné ke zaznamenávání chybových stavů, čehož využíváme při propagaci chyb zpět rekurzivním sestupem.

Další velkou částí našeho projektu je **sémantický analyzátor**, na kterém pracoval hlavně David. Pracuje ve 4 fázích. Během parsingu se do rámců přidávají položky a kontrolují se nedefinované a vícekrát definované proměnné. Po parsingu se kontroluje přítomnost funkce main() a správnost prologu. Následně se kontrolují těla funkcí, jejich příkazy a výrazy, probíhá rekurzivní průchod stromem. Nakonec se prohledávají všechny rámce programu a kontroluje se využití proměnných a neměnnost konstant. Našimi rozšířeními jsou kontrola, že se příkaz return u funkcí s návratovou hodnotou vyskytuje v každé větvi těla této funkce; detekce mrtvého kódu za příkazem return; implicitní konverze při přiřazení; vyhodnocování výrazů se známou hodnotou při překladu.

Poslední částí je **generátor kódu**, na kterém pracovali hlavně Lukáš a David. Generátor prochází stromem podobně jako sémantický analyzátor. Pro zajištění unikátních identifikátorů je proměnným v mezikódu přidáván sufix s číslem jejich rámce. Při zanoření do cyklu se nejdříve definují všechny proměnné v jeho těle, až poté se generuje podmínka s tělem cyklu, a to již bez definic. Tímto zamezujeme redefinici proměnných v těle cyklu. Při vyhodnocování výrazů je výsledná hodnota předána na vrchol zásobníku a všechny operace dále používají zásobníkovou verzi, pokud je to možné. V rámci optimalizace se snažíme sekvenci instrukcí vestavěných funkcí vkládat přímo do těla volající funkce, čímž se sníží počet volání funkcí a tím i výpočetní čas. Pro optimalizaci se instrukce PUCHS a POPS předávají do optimalizačního bufferu, který se je pokusí nahradit instrukcí MOVE. Díky této optimalizaci došlo k redukci instrukcí přibližně o 15 %.

Na závěr ještě dodám, že náš projekt jsme také rozsáhle **testovali**. Pro každou oddělenou část jsme psali vlastní testy. Nakonec jsme vytvořili i integrační testy, se kterými jsme testovali spolupráci mezi oddělenými moduly, a tedy chod celého překladače jako jednoho celku.

Takže vám děkuji za pozornost a na rozloučenou tady máte **shrnutí našich vlastních rozšíření**, práce nad rámec zadání a dalších specialit.