FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Dokumentace k projektu z předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ17

Tým 83, varianta I

6. prosince 2017

Rozšíření
SCOPE
GLOBAL
UNARY
BASE
FUNEXP
IFTHEN

Ondřej Pavela	**xpavel34**	**25%**
Richard Gall	xgallr00	25%
Lukáš Letovanec	xletov00	25%
Robert Navrátil	xnavra61	25%

Obsah

1 Úvod

V následujících kapitolách této projektové dokumentace bude popsáno, jakým způsobem se náš tým postavil k řešení rozsáhlého projektu, kterým překladač jazyka IFJ17 bezesporu je. Dokumentace je rozdělena do několika kapitol, ve kterých se postupně věnujeme autorskému návrhu a realizaci jednotlivých části celého překladače.

Úvodní kapitola se zaměřuje na organizační problémy, které jsou při práci ve větším týmu nevyhnutelné a na jejichž zvládnutí často závisí osud celého projektu. Druhá kapitola se věnuje nejprve obecnému návrhu překladače jako celku s důrazem na kvalitní návrh stěžejních datových struktur a rozhraní mezi hlavními celky překladače. V jednotlivých podkapitolách se poté členové týmu zabývají návrhem a implementací konkrétních částí, použitými algoritmy a překážkami, jež bylo nutno překonat.

Závěr čtenáři představuje kritické zhodnocení výsledného produktu, ve kterém se jakožto vedoucí týmu zabývám tím, co se z mého osobního pohledu v projektu zdařilo, jaké zkušenosti jsme si jako tým z projektu odnesli a co bychom retrospektivně řešili jinak.

V příloze se poté nachází diagram konečného automatu, který reprezentuje náš lexikální analyzátor. Dále je v příloze obsažena specifikace LL-gramatiky, LL-tabulka, redukční pravidla výrazů a precedenční tabulka jejichž kombinace tvoří jádro celého překladače.

2 Práce v týmu

2.1 Rozdělení prací

Jednou z hlavních priorit pro nás bylo včasné dokončení celého projektu, abychom mohli projekt pokusně odevzdat a zároveň měli dostatek času výsledek otestovat a vyladit. Proto jsme uskutečnili první týmovou schůzku již druhý týden semestru jakmile jsme měli finální sestavu týmu.

Na schůzce jsme se nezabývali konkrétními implementačními detaily jednotlivých částí, jelikož někteří členové dopředu nevěděli, která část překladače jim bude svěřena a neměli v danou chvíli dostatečnou znalost problematiky. Místo toho jsme se zaměřili na vytvoření základní společné představy o tom, jak by výsledné části překladače spolu mohly komunikovat, tedy jak by mohla vypadat rozhraní stěžejních modulů.

Po získání této představy jsme na základě společné domluvy rozdělili práci mezi členy týmu víceméně dle přání každého z nás. **Richard Gall** dostal na starost vytvoření konečného automatu pro účely lexikální analýzy. **Robert Navrátil** si vybral jako svůj předmět zájmu precedenční analýzu v rámci syntaktické analýzy. **Lukáš Letovanec** projevil zájem o syntaktickou analýzu shora dolů, kterou jsem si však chtěl jako vedoucí týmu vzít na starost, jelikož se jedná o samotné srdce syntaxí řízeného překladu, na kterém závisí všechny zbylé moduly. Po vzájemné dohodě byla **Lukáši** přidělena závěrečná, neméně důležitá, fáze překladu - generátor kódu. Já (**Ondřej Pavela**) jsem si tedy vzal na starost již zmíněnou syntaktickou analýzu shora dolů. Jelikož jsem však cítil jistou zodpovědnost za případný úspěch či neúspěch celého projektu, rozhodl jsem se vzít si na starost také vytvoření pomocných datových struktur. S některými z těchto struktur ve výsledku pracovaly všechny části překladu a bylo tedy velice důležité je kvalitně navrhnout a implementovat. Jedná se o tabulku symbolů, strukturu pro tokeny a zásobník pro obě části syntaktické analýzy. Bližší popis těchto struktur a jejich rozhraní bude poskytnut v následující kapitole.

Toto rozdělení práce se nám ve výsledku podařilo dodržet s jedinou výjimkou. Z důvodu menší časové tísně se **Robert Navrátil** postaral o napsání podprogramu pro generování mezikódu výrazů v rámci generátoru mezikódu jako celku.

2.2 Schůzky, komunikace a časový plán

Na úvodním setkání jsme usoudili a po společné domluvě odsouhlasili, že není nezbytně nutné provádět týmové schůzky pravidelně každý týden, jelikož by se jednalo o zbytečnou časovou zátěž. Místo toho jsme se snažili plánovat si další týmová setkání vždy při příležitosti dosažení některého z vývojových milníků.

Při zpětném ohlédnutí to považuji za správné rozhodnutí, které ovšem mělo i svá negativa. Spoléhali jsme na samostatnost a zodpovědnost jednotlivých členů, a proto měl každý z nás naprosto volnou ruku co se týče návrhu a implementace své částí. Jedinou výjimku představovala předem dohodnutá rozhraní, jenž měl každý z nás povinnost dodržet. Díky tomuto přístupu bylo možné redukovat počet schůzek a věnovat tak více času samotné práci. Nepříjemným vedlejším efektem se ovšem stala snížená orientace v cizím kódu a obecně menší přehled o tom, jak pracují moduly ostatních členů týmu. Při opravách chyb v závěrečných fázích vývoje bylo proto nutné delegovat práci na opravách členům, v jejichž části se chyba nacházela.

Největším prohřeškem naší práce v týmu se tak z mého pohledu stalo nevyužití techniky *code review*. Bez ní byla čitelnost kódu často na špatné úrovni a zároveň bylo těžší poskytovat si navzájem zpětnou vazbu, která by měla pozitivní dopad na kvalitu kódu.

Jako hlavní komunikační kanál jsme zvolili Facebook. Pro společnou komunikaci jsme vytvořili

konverzační skupinu, kde se debatovalo o týmových a organizačních otázkách. Problémy a chyby spojené s konkrétní částí překladače se poté rozebíraly v privátních konverzacích se zodpovědnou osobou. Vzájemná nedorozumění či nejasnosti v zadání jsme se snažili řešit osobní domluvou mimo schůzky, jelikož se tři členové týmu potkávali denně ve škole.

Časový plán prací našeho týmu nebyl založen na striktních předem určených termínech dokončení jednotlivých částí. Cílem naopak bylo dokončit projekt v co nejkratším čase, a proto jsme začali pracovat ihned po zveřejnění zadání.

Již za dva týdny jsme měli k dispozici hotové všechny pomocné struktury, které se s časem drobně upravovaly vzhledem k měnícím se požadavkům. Kromě toho se nám podařilo vytvořit prvotní verzi syntaktického analyzátoru bez zpracování výrazů. Díky vhodně navrženému rozhraní mezi lexikálním a syntaktickým analyzátorem bylo možné poměrně jednoduše testovat funkčnost syntaktické analýzy i bez existujícího lexikálního analyzátoru, čehož jsme samozřejmě využili.

V této fázi jsme uskutečnili další týmovou schůzku, na které byli všichni členové obeznámeni s aktuálním stavem jednotlivých modulů a projektu jako takového. Z důvodu nedostatku času jsme se ovšem nezabývali čtením kódu a místo toho proběhla debata, jak dále postupovat. Ve výsledku byl urgován **Richard Gall**, aby co nejdříve vytvořil lexikální analyzátor, jelikož se jednalo o další logický krok ve vývoji. Kromě toho začal pracovat **Robert Navrátil** na precedenční analýze, kterou jsme se chystali integrovat do již existujícího rámce syntaktické analýzy. **Ondřej Pavela** začal pracovat na sémantických kontrolách v rámci analýzy shora dolů. Na úvodní schůzce bylo totiž rozhodnuto, že z hlediska jednoduchosti bude nejpřínosnější tyto kontroly těsně provázat se syntaktickou analýzou. **Lukáš Letovanec** byl bohužel v časové tísni a neměl tak možnost začít pracovat na generátoru kódu, což však nebyl v dané chvíli až takový problém.

Během následujících tři týdnů se nám podařilo vytvořit funkční lexikální analyzátor, dokončit syntaktický analyzátor včetně sémantických kontrol a obě části úspěšně integrovat. Od té chvíle již bylo možné důkladněji testovat překladač pomocí skutečného vstupu, což nám umožnilo odhalit a opravit poměrně velké množství chyb v jejich počátku.

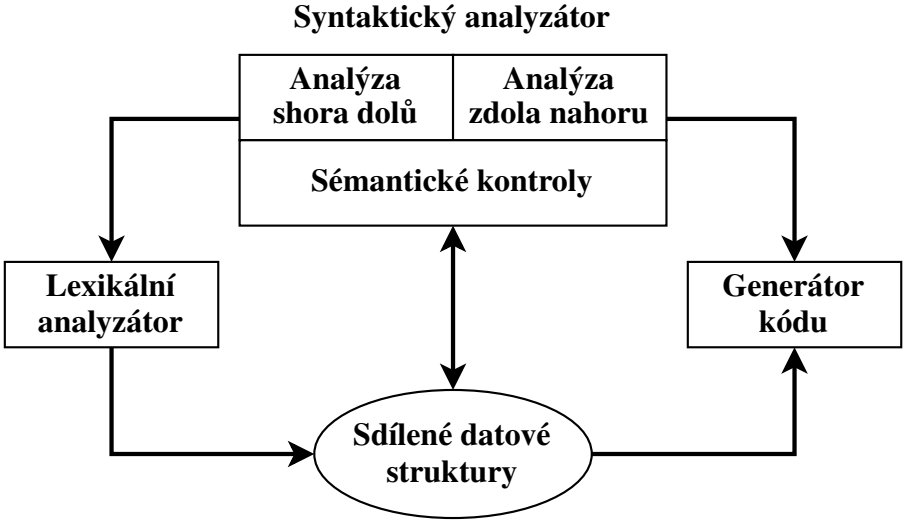
Poslední nedokončenou části zůstával generátor kódu, který měl na starost **Lukáš**. Ten se však delší dobu nemohl zapojit do týmových prací z důvodu časové vytíženosti, a tak projekt ustrnul na mrtvém bodě. V mezičase se proto zbytek týmu zabýval testováním a opravami chyb. S postupem času se však zvyšovala naše nervozita, a proto jsme začali studovat teorii ohledně generování kódu pro případ, že by **Lukáš** nestihl generátor vytvořit včas. K tomu naštěstí nedošlo a s pomocí **Roberta** se podařilo kompletní neodladěnou verzi překladače dokončit týden před pokusným odevzdáním. Ve zbývajícím čase jsme se naplno věnovali testování a opravám chyb, abychom byli schopni v termínu odevzdat finální a odladěnou verzi.

2.3 Sdílení a správa zdrojového kódu

Na úvodní schůzce jsme se shodli na nezbytnosti použití nějakého z existujících řešení pro správu zdrojového kódu. U projektů většího rozsahu je totiž zásadní, aby bylo sdílení změn s ostatními členy týmu rychlé a efektivní. Na základě předchozí zkušenosti všech členů jsme proto zvolili verzovací systém *Git*. Jako datové úložiště jsme zvolili uživatelsky přívětivý *Bitbucket*, který poskytuje možnost vytvoření privátního repozitáře zdarma.

3 Návrh a implementace

Cílem projektu bylo vytvořit překladač pro jazyk IFJ17, který vychází z jazyka FreeBASIC. Vzhledem k rozsahu celého projektu bylo nutné nějakým vhodným způsobem dekomponovat problém na dílčí části tak, aby mohl celý tým pracovat pokud možno souběžně. Po kratší úvaze jsme se proto rozhodli rozdělit překladač na následující navzájem komunikující části:



Obrázek 1: Základní dekompoziční koncept

Dále bylo potřeba vymyslet vhodná rozhraní, skrze která by probíhala veškerá komunikace mezi jednotlivými celky.

Pro komunikaci mezi syntaktickým a lexikálním analyzátorem jsme se rozhodli vytvořit prostředníka v podobě modulu, pojmenovaném podle sdílené datové struktury pro tokeny. Tento krok nám umožnil v prvopočátcích simulovat vstup pro syntaktický analyzátor, jak již bylo zmíněno dříve.

Interní komunikace mezi prediktivní a precedenční analýzou v rámci syntaktického analyzátoru byla navržena v duchu jednoduchosti. Jakmile prediktivní analýza narazí na neterminál oznamující začátek výrazu, předá řízení precedenčnímu syntaktickému analyzátoru. Ten je na základě obdržené informace o předchozím tokenu schopen určit, jaké druhy výrazů jsou v daném kontextu korektní. Po zpracování výrazu se předá řízení zpět prediktivní analýze spolu s informací o výsledném typu celého výrazu, aby bylo možné provést potřebné sémantické kontroly.

Rozhraní mezi syntaktickým analyzátorem a generátorem kódu je již o něco složitější. V průběhu syntaktické analýzy se jednotlivé načtené tokeny a aplikovaná pravidla vkládají do dvou bufferů. Po každém zpracovaném řádku zdrojového kódu se předá řízení generátoru, který na základě dat uložených v bufferech vygeneruje mezikód pro daný řádek kódu a na závěr vyprázdní oba buffery. Tento cyklus se opakuje tak dlouho, dokud syntaktický analyzátor nenarazí na konec vstupu.

Kromě rozhraní jsme se předem domluvili na dodržování určitých pravidel, vytvořených zejména za účelem minimalizace úniků paměti. Základ představovala zodpovědnost každého modulu za správu vlastní paměti. Ve výsledku tak každý modul splňuje požadavek, že všechna jím alokovaná paměť je korektně uvolněna jeho vlastní funkcí v případě ukončení programu (s chybou i bez). Dalším požadavkem bylo, aby všechny ukazatele na dynamicky alokovanou paměť byly uloženy v globálních proměnných nebo jednoduchých strukturách. Tímto jsme zamezili časté programátorské chybě, kdy se ukazatel na alokovanou paměť nějakým nedopatřením ztratí, například při návratu z funkce. Zbylá pravidla nebyla až tak podstatná a spíše sloužila jako doporučení pro jednotlivé členy týmu, jakým

způsobem s pamětí pracovat. Snažili jsme se především minimalizovat počet alokací, jelikož se jedná o pomalou operaci. Proto se nejčastěji využívalo struktur, které sloužily pro uložení ukazatelů na větší bloky paměti spolu s metadaty o jejich využití. Tento způsob nám zároveň umožnil opakované použití již alokované paměti bez nutnosti jejího uvolnění a opětovné alokace.

Posledním požadavkem bylo dodržování stanovené štábní kultury při psaní kódu, aby se zamezilo nejednotnému stylu v rámci celého projektu.

3.1 Sdílené datové struktury

3.1.1 Struktura pro tokeny

Struktura tokenu se skládá pouze z pěti datových položek. V položce `value`, typu `void**`, je uložen ukazatel na konkrétní data spojená s daným lexémem (literál, název proměnné, nebo textová reprezentace klíčového slova). Samotná data jsou však uložena v pomocné globální struktuře, kvůli snížení rizika úniku paměti.

Důvod, proč je první položka typu `void**` je vcelku prostý. Všechny druhy literálů a názvy proměnných jsou uloženy v několika velkých dynamicky alokovaných polích, které se jednou za čas mohou zvětšit při nedostatku paměti pro uložení dat. Při realokaci ovšem může dojít k přemístění dat, a proto je v položce `value` uložen pouze ukazatel na počátek pole. Samotný ukazatel ovšem nestačí, což řeší další datová položka struktury, jejíž hodnota reprezentuje offset konkrétních dat od počátku pole. Dále je ve struktuře uložena informace o tom, jaký terminál reprezentuje hodnotu tokenu. Poslední položkou je obecný typ tokenu pro jednodušší porovnávání.

V hlavičkovém souboru se nachází pouze deklarace datového typu a číst či modifikovat jednotlivé tokeny tak lze jedině skrze funkce rozhraní. Tímto jsme v rámci možností jazyka `C` zapouzdřili celou strukturu a zbytek týmu se tak již nemusel zabývat konkrétní implementací.

3.1.2 Tabulka symbolů

Náš tým si vybral variantu zadání, ve které se měla tabulka symbolů implementovat pomocí binárního vyhledávacího stromu. Při návrhu jsme zároveň již počítali s pozdější implementací rozšíření SCOPE a GLOBAL, což mělo velký vliv na finální podobu této datové struktury.

Tuto základní datovou strukturu jsme se však rozhodli vhodným způsobem modifikovat pomocí hashování klíčů. Motivací pro tuto modifikaci se stala pomalá a neustále se opakující operace porovnávání potenciálně velmi dlouhých řetězců. Využili jsme hashovací funkce `djb2` od *Dana Bernsteina*[1], která údajně poskytuje velmi dobré výsledky. Tato modifikace ovšem vedla na problém s možnou kolizí dvou klíčů, a proto jsme dále upravili strukturu pro uzly stromu tak, aby podporovala jejich zřetězení do jednosměrně vázaného seznamu.

Rozhraní pro práci s tabulkami ve výsledku obsahuje pouze funkce pro vkládání a vyhledávaní uzlů. Usoudili jsme, že vytvářet funkci pro mazání jednoho uzlů je zbytečné, jelikož by zůstala nevyužitá a pro mazání celého stromu by navíc byla neefektivní.

Na závěr bylo potřeba vymyslet vhodný způsob, jakým organizovat více instancí tabulek symbolů pro potřeby rozšíření. Jedna konkrétní instance představuje globální tabulku pro ukládání identifikátorů funkcí a globálních proměnných spolu s dodatečnými důležitými informacemi. Všechny ostatní tabulky slouží pro ukládání lokálních symbolů, přičemž každá tabulka obsahuje kromě svých dat také ukazatel na svého rodiče, což umožňuje simulovat libovolné zanořování bloků. Při vyhledávání konkrétního lokálního symbolu poté postupujeme směrem vzhůru v hierarchii tabulek,

[1] Zdroj: http://www.cse.yorku.ca/ oz/hash.html

dokud nenalezneme hledaný symbol, nebo dokud nenarazíme na vrchol hierarchie, což představuje neúspěšné vyhledání.

Při návrhu rozhraní byl zároveň kladen požadavek na jednoduchost použití z pozice koncového uživatele modulu. Proto jsme vytváření tabulek a pohyb v jejich hierarchii opět zapouzdřili. Výsledné rozhraní zpřístupňuje pouze tři funkce: `BeginSubScope`, `EndSubScope` a `EndScope`. Ty se starají o provedení veškerých potřebných akcí spojených se zanořováním a vynořováním z bloků, přičemž samotný modul si interně pamatuje aktivní tabulku.

3.1.3 Zásobník symbolů

Poslední jednoduchou strukturou je zásobník symbolů. Slouží pro ukládání terminálů a neterminálu, se kterými pracuje syntaktický analyzátor a zároveň je jako takový potřeba pro realizaci algoritmů v rámci analýzy. Opět jsme se pokusili celý zásobník zapouzdřit a zpřístupnit pouze nezbytné funkce rozhraní. Koncový uživatel datové struktury se tak nemusí zabývat správou paměti, která je zcela v režii modulu.

Rozhraní obsahuje jak klasické operace nad zásobníkem (`Push`, `Pop` a `Top` v různých variantách), tak různé specializované funkce pro účely analýzy zdola nahoru. Poslední zajímavou funkcí je `ExpandTop`, která provádí derivaci neterminálu na vrcholu zásobníku. Využije předaného terminálu a pokusí se vyhledat odpovídající derivační pravidlo v LL tabulce. Jestliže dané pravidlo existuje, provede derivaci a vrátí identifikátor pravidla. V opačném případě vrátí hodnotu signalizující neúspěch, což vede k ukončení překladu.

3.2 Lexikální analýza

Stavový automat je implementován za pomocí výčtového typu `State`. V samotném programu jsou jednotlivé stavy vybírány pomocí konstrukce `switch`. Uvnitř každého stavu je také interní logika zpracovávající podmínky, aby nebylo potřeba vytvářet další stavy. Ta také nastavuje příznaky jednotlivých stavů a sleduje počet znaků v aktuálním lexému.

Užitím poněkud neobvyklé techniky lexikální analýzy, tj. nejprve se provede lexikální analýza celého zdrojového souboru a až poté následuje syntaktická analýza, se v případě lexikální chyby na místo okamžitého ukončení kompilátoru, vytvoří token typu `UNDEFINED`. Tímto předcházíme ukončení programu se špatným návratovým kódem v případě, že se ve zdrojovém kódu dříve objevil jiný druh chyby.

Tokeny jsou generovány ve funkci `Lexical()` a následně uloženy do pole ve sdílené datové struktuře viz. kapitola Sdílené datové struktury.

Lexikální analýza interně rozlišuje dva typy tokenů: short a long. Typ short značí token, který je tvořen jedním znakem (např ';' nebo '='). Oproti tomu typ long značí token složený z více znaků. Při ukončení lexému znakem, který není jednoznakým lexémem, či bílým znakem, bude aktuální stav změněn za pomocí funkce `SetState()`.

Výčtový typ `Type` obsahuje hodnoty, kterých může nabývat příznak `endFlag`, který určuje, zda došlo k ukončení aktuálního lexému, či začátku nového. Vyhodnocení `endFlag` probíhá ve funkci `IsEnd`.

3.3 Syntaktická analýza

3.3.1 Analýza shora dolů

Analýza shora dolů je ve své podstatě velmi jednoduchá. Náš tým se však rozhodl implementovat prediktivní variantu místo rekurzivního sestupu, což vyžadovalo vytvoření další datové struktury pro fyzickou reprezentaci LL tabulky.

V původní verzi byla LL tabulka implementována naivně jako pouhé dvojrozměrné pole s konstantní složitostí u operace vyhledání pravidla. Později se však ukázalo, že tento způsob implementace má více negativ než pozitiv a to obzvláště v období bouřlivého vývoje, kdy se často zasahovalo do samotné gramatiky, ať už kvůli chybám či rozšířením. Proto jsme se později rozhodli LL tabulku přepracovat i za cenu ztráty rychlosti spojené s konstantní složitostí vyhledání pravidla. Výsledná „tabulka“ je implementována jako pole dvojic pro každý neterminál v naší navržené gramatice. Každá dvojice je poté složena z terminálu a odpovídajícího derivačního pravidla, které je spojeno s danou kombinací přečteného terminálu a neterminálu z vrcholu zásobníku. Jinými slovy má každý neterminál přiřazen svůj vlastní slovník, kde terminál je klíčem a pravidlo hodnotou.

Neterminál N	
Terminál 1	Pravidlo A
Terminál 2	Pravidlo B
Terminál 3	Pravidlo ε
...	

Obrázek 2: Slovník neterminálu

Při hledání derivačního pravidla se poté sekvenčně (jsme si vědomi neefektivity) prochází konkrétní slovník. Pokud slovník neobsahuje hledaný klíč, dochází k derivační chybě (neexistující pravidlo).

Samotnou analýzu jsme implementovali pomocí algoritmu, který byl prezentován na přednáškách, a proto zde nebude popisován. Spíše se zaměříme na jeho modifikace oproti přednášené verzi. Jelikož jsme se rozhodli neimplementovat sémantický analyzátor jako samostatnou část našeho překladače, bylo nutné nějakým způsobem provázat sémantické kontroly s jednotlivými kroky syntaktické analýzy. V našem podání je princip následující: po každém úspěšném porovnání načteného a očekávaného terminálu se provedou dodatečné logické (zanoření, vynoření, apod.) a sémantické (kontrola existence identifikátorů, kompatibilita typu, apod.) akce spojené s konkrétním terminálem. Obdobný princip je aplikován při derivaci neterminálů, kde máme ke konkrétnímu neterminálu opět přidruženy dodatečné akce.

Dále bylo potřeba provádět některé syntaktické kontroly až po úspěšném porovnání terminálů a to kvůli několika problematickým rysům jazyka IFJ17. Jedním z nich je například podmínka, že příkaz návratu z funkce je možné použít pouze v definici funkce. Tento problém je samozřejmě řešitelný i na úrovni gramatiky, což by však vedlo k dramatickému navýšení celkového počtu pravidel. Zmíněné řešení je dle nás daleko elegantnější a jednodušší.

Jelikož je náš překladač založen na konceptu syntaxí řízeného překladu, tak mezi logické akce spojené s konkrétními terminály patří samozřejmě i volaní podprogramu pro generování mezikódu. Princip rozhraní byl popsán v úvodu kapitoly, tak jej zde nebudeme opakovat. Rádi bychom zde však prezentovali ukázkovou chybu návrhu, které jsme se dopustili.

V původní verzi se zmíněný podprogram pro generování mezikódu volal vždy po zpracování určitého bloku, do kterých jsme zdrojový kód členili. Bohužel jsme si neuvědomili, že při použití této techniky bude docházet k chybám spojeným s viditelností proměnných. Pokud se v části bloku odkazujeme na proměnnou z nadřazeného bloku kódu a v další části je poté tato proměnná překryta nově definovanou proměnnou stejného jména, dojde k vygenerování nekorektního mezikódu. To má samozřejmě návaznost na způsob, jakým je implementována tabulka symbolů. Při volání generátoru již lokální tabulka symbolů obsahovala později definovanou lokální proměnnou stejného jména, přičemž funkce pro vyhledání symbolu v tabulce symbolů navrací první nalezenou shodu. Při generování kódu jsme se tak odkazovali na jinou proměnnou stejného jména.

Tento problém jsme nakonec vyřešili jednoduše tak, že se ve finální verzi generuje po jednotlivých řádcích zdrojového kódu. Bohužel jsme však ztratili čas psaním kódu, který jsme nakonec stejně zahodili a to vše jen kvůli nedomyšlenému návrhu.

3.3.2 Analýza zdola nahoru

V rámci zpracování výrazů bylo třeba nejprve vhodně navrhnout precedenční tabulku a vymyslet redukční pravidla, která by správně zpracovávala posloupnosti neterminálů a terminálů nacházejících se na zásobníku. Po návrhu těchto komponent se přešlo k samotné implementaci. Tyto dvě struktury se v průběhu vývoje často upravovaly s přibývajícími požadavky a také v důsledku naivity při tvorbě počátečního návrhu.

V průběhu implementace nastalo mnoho překážek a komplikací. Jeden z největších problémů se projevil po rozhodnutí implementovat rozšíření FUNEXP, které přidává podporu pro zápis výrazů v argumentech funkcí. Jelikož jsou výrazy zpracovávány klasicky zleva doprava, tak by jejich přítomnost nehrála velkou roli, ale následné volání dalších funkcí v parametrech způsobovalo značné komplikace. Tímto vyvstaly problémy s kontrolováním počtu parametrů. Řešení jsme nalezli v samostatné funkci, která je volána pouze pokud je příchozí token identifikátor značící funkci. Jestliže narazí na volání funkce, tak se pouze rekurzivně zavolá a globální proměnná s počtem zanoření se inkrementuje. Takto se jednoduše uchovává index aktuálního parametru.

Další nepříjemností se stalo rozšíření UNARY. Původní návrh byl jednoduchý, ale změna priority při přítomnosti operátoru s vyšší prioritou před samotným operátorem mínus přinesla nepříjemné komplikace. Dále se k tomuto také váže zvláštnost jazyka FreeBasic, který podporuje zápis více po sobě jdoucích unárních operátorů. Z tohoto také vyplývá, že všechny zápisy typu $--6$ $--5$ apod. jsou validní.

3.4 Generování kódu

Generátor výsledného mezikódu IFJcode17 zpracovává jednotlivé pravidla, které mu předává syntaktický analyzátor vždy po ukončení analýzy jednoho řádku zdrojového kódu. Generátor následně v cyklu projde buffer s uloženými pravidly a pro každé z nich vygeneruje odpovídající tříadresnou, nebo zásobníkovou instrukci, popřípadě zavolá podprogram pro zpracování výrazů. Vygenerované instrukce jsou ukládány do jednoho ze dvou bufferů, ze kterých je ve výsledku složen celkový výstup překladače. Dva buffery slouží pro simulaci datové a kódové oblasti programu zapsaného v mezikódu.

V datové oblasti se nachází instrukce pro deklaraci a případnou inicializaci globálních, statických a pomocných interních proměnných. Dále se zde nachází také instrukce pro deklaraci proměnných, které byly ve zdrojovém kódu deklarovány uvnitř nějakého cyklu. Buffer kódové oblasti slouží pro uložení instrukcí uživatelských funkcí a vestavěných funkcí jazyka IFJ17, o jejichž implementaci se stará generátor. Samozřejmě slouží také pro ukládání instrukcí hlavního těla programu.

Během řešení jednotlivých problému spojených s generováním mezikódu jsme se zdrželi zejména u generování vhodných návěští pro podmíněné příkazy. Dlouho jsme nemohli vymyslet vhodný způsob a celý problém se zkomplikoval zejména kvůli rozšíření SCOPE. Nakonec jsme přišli s technikou perzistentních a jednorázových návěští, což se ukázalo být vhodným řešením tohoto problému. Tato technika byla použita pro korektní párování návěští s odpovídajícími skoky a je založena na použití dvou zásobníku. Do jednoho ze zásobníků vkládáme prvky pouze pokud narazíme na začátek podmíněného příkazu (`IF`) a vybíráme prvky pouze pokud narazíme na konec poslední větve podmíněného příkazu (`END IF`). U druhého zásobníku vkládáme a vybíráme prvky při zpracování libovolné větvě podmíněného příkazu. Na základě počtu prvků v obou zásobnících jsme poté schopni rozhodnout, zda v některých případech generovat vybranou instrukci či nikoliv.

Použitím této techniky jsme se bohužel nevyhnuli generování redundantních návěští v některých krajních případech, což však nemá na výslednou funkčnost programů žádný vliv.

3.4.1 Generování kódu výrazů

Podprogram pro generování výrazů je volán z funkce generující mezikód IFJcode17. Samotné generování výrazů nebylo příliš složité. Mírné komplikace nastaly až při implicitních konverzích. Určité konverze bylo nutné řešit mimo tento podprogram přímo v generátoru. Těmito výjimkami jsou speciální přiřazovací operátory $+=$, apod. Dále také RETURN, u kterého není jasný výsledný typ. V závěru jsme narazili na nejasnosti u celočíselného dělení, které jsme chápali jako operující pouze s celými čísly, přičemž jsme tedy zakazovali implicitní konverze z DOUBLE na INTEGER.

4 Závěr

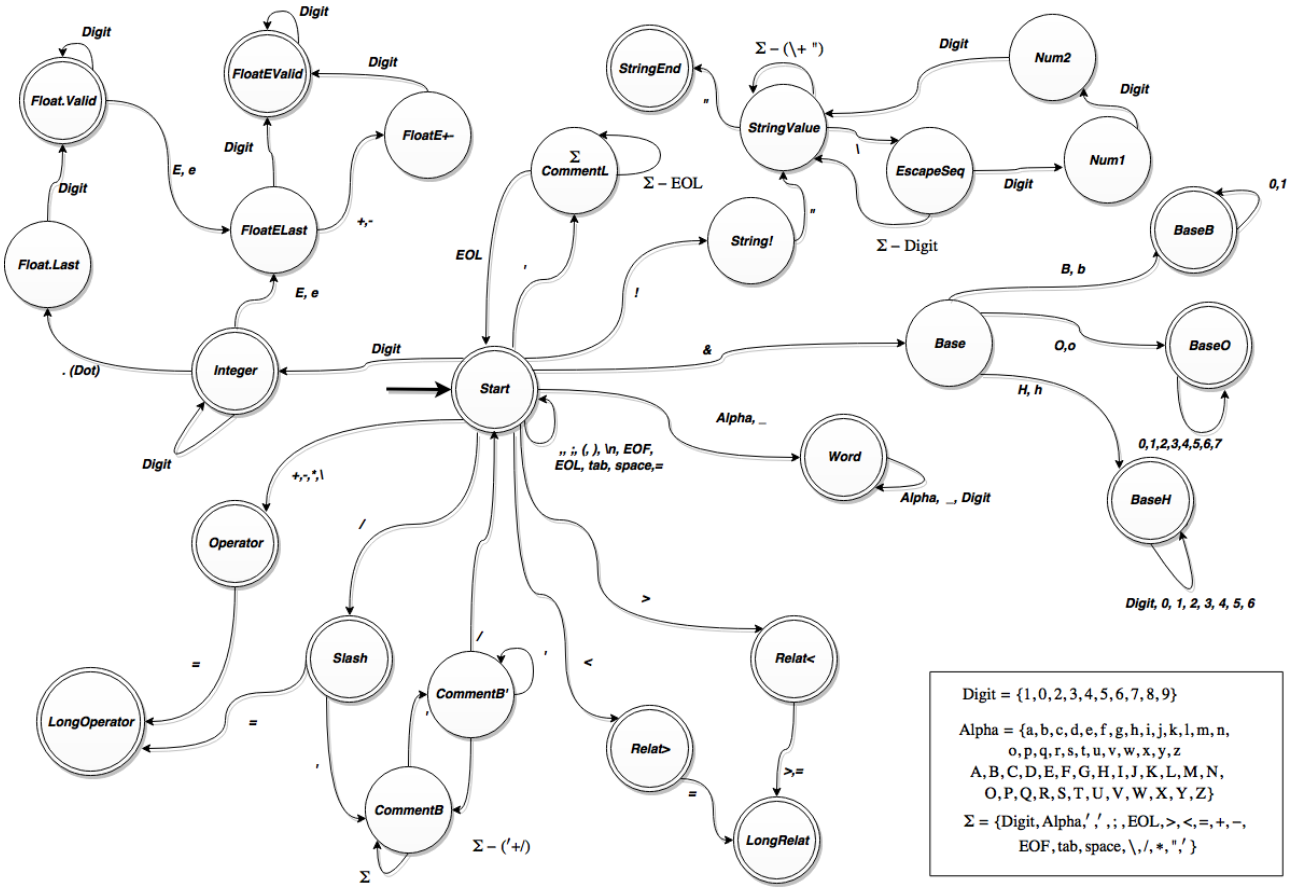
Vytvořit vlastní překladač pro nás bylo největší a nejzajímavější výzvou dosavadního studia na fakultě. Kvalitně navrhnout architekturu a posléze vlastní návrh převést na konkrétní implementaci vyžadovalo skloubení mnoha dovedností. Nestačilo pouhé porozumění přednášené teorii, naopak se od každého studenta očekávala dovednost tuto teorii prakticky aplikovat. Neméně podstatným předpokladem úspěchu byla samozřejmě schopnost kooperace ve větším týmu, což bylo něco, s čím jsme prozatím neměli zkušenost. Práce vyžadovala neustálou týmovou komunikaci a zvládnutí technických překážek v podobě sdílení kódu a psaní vlastních testů.

Při zpětném pohledu na odvedenou práci jsme jako tým s naším výsledným produktem relativně spokojení. Jistě je zde mnoho ne úplně zdařilých řešení, neoptimálních algoritmů a nedostatků v návrhu jednotlivých částí, my však věříme, že tím nejhodnotnějším, co si z tohoto projektu odnášíme, neměl být výsledný překladač, ale nabyté zkušenosti do dalších projektů a především praxe.

Výsledný překladač byl vyvíjen a testován na operačních systémech Windows 10 a Ubuntu 16.04 se 64-bitovou architekturou. Očekává se však bezproblémová funkčnost i na systémech s 32-bitovou architekturou vzhledem k obecnému návrhu a implementaci.

5 Přílohy

5.1 Automat lexikální analýzy



Obrázek 3: Konečný automat bez chybových stavů

5.2 LL gramatika

1: **PROG**	→	**SCOPE LINE-BREAK**
2: **PROG**	→	declare **HEADER PROG**
3: **PROG**	→	**FUNC PROG**
4: **PROG**	→	dim shared *ID* as **TYPE INIT** *EOL* **LINE-BREAK PROG**
5: **SCOPE**	→	scope *EOL* **LINE-BREAK STAT-LIST** end scope
6: **HEADER**	→	function *ID*(**ARG**) as **TYPE** *EOL* **LINE-BREAK**
7: **FUNC**	→	**HEADER STAT-LIST** end function *EOL* **LINE-BREAK**
8: **ARG**	→	*ID* as **TYPE NEXT-ARG**
9: **ARG**	→	ε
10: **NEXT-ARG**	→	**, ARG**
11: **NEXT-ARG**	→	ε
12: **STAT-LIST**	→	**STAT** *EOL* **LINE-BREAK STAT-LIST**
13: **STAT-LIST**	→	ε
14: **STAT**	→	dim *ID* as **TYPE INIT**
15: **STAT**	→	static *ID* as **TYPE INIT**
16: **STAT**	→	scope *EOL* **LINE-BREAK STAT-LIST** end scope
17: **STAT**	→	*ID* **OP EXPR**
18: **STAT**	→	if **EXPR** then *EOL* **LINE-BREAK STAT-LIST ELSEIF ELSE**
19: **STAT**	→	do while **EXPR** *EOL* **LINE-BREAK STAT-LIST** loop
20: **STAT**	→	input *ID*
21: **STAT**	→	print **EXPR ; NEXT-EXPR**
22: **STAT**	→	return **EXPR**
23: **ELSEIF**	→	elseif **EXPR** then *EOL* **LINE-BREAK STAT-LIST ELSEIF**
24: **ELSEIF**	→	ε
25: **ELSE**	→	else *EOL* **LINE-BREAK STAT-LIST** end if
26: **ELSE**	→	end if
27: **NEXT-EXPR**	→	**EXPR ; NEXT-EXPR**
28: **NEXT-EXPR**	→	ε
29: **INIT**	→	= **EXPR**
30: **INIT**	→	ε
31: **TYPE**	→	integer
32: **TYPE**	→	double
33: **TYPE**	→	string

Tabulka 1: Derivační pravidla 1/2

34:	**OP**	→	=
35:	**OP**	→	+=
36:	**OP**	→	−=
37:	**OP**	→	*=
38:	**OP**	→	\=
39:	**OP**	→	/=
40:	**LINE-BREAK**	→	*EOL* **LINE-BREAK**
41:	**LINE-BREAK**	→	ε

Tabulka 2: Derivační pravidla 2/2

5.3 LL tabulka

	PROG	SCOPE	HEADER	FUNC	ARG	NEXT-ARG	STAT-LIST	STAT	ELSEIF	ELSE	NEXT-EXPR	INIT	TYPE	OP	LINE-BREAK	$
As																
Asc																
Declare	2														41	
Dim	4						12	14							41	
Do							12	19							41	
Double											27		32			
Else							13		24	25					41	
End							13		24	26					41	
Chr																
Function	3		6	7											41	
If							12	18							41	
Input							12	20							41	
Integer											27		31			
Length																
Loop							13								41	
Print							12	21							41	
Return							12	22							41	
Scope	1	5					12	16							41	
String											27		33			
SubStr																
Then																
While																
Elseif							13		23						41	
Shared																
Static							12	15							41	
ID					8		12	17			27				41	
EOL											28	30			40	
EOF															41	
,						10										
;																
(											27					
)					9	11										
-											27					
=												29		34		
+=														35		
-=														36		
*=														37		
\=														38		
/=														39		
$																END

Tabulka 3: LL tabulka

5.4 Precedenční tabulka

Použitá precedenční tabulka:

	+	−	*	/	\	<	<=	>	>=	<>	=	(	)	i	f	,	s	$
+	>	>	<	<	<	>	>	>	>	>	>	<	>	<	<	>	<	>
−	>	>	<	<	<	>	>	>	>	>	>	<	>	<	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	>	<	>
\	>	>	<	<	>	>	>	>	>	>	>	<	>	<	<	>	<	>
<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<		<	>
<=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<		<	>
>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<		<	>
>=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<		<	>
<>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<		<	>
=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<		<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	chyba / <	<	
)	>	>	>	>	>	>	>	>	>	>	>		>			>		>
i	>	>	>	>	>	>	>	>	>	>	>		>			>		>
f												=						
,	<	<	<	<	<							<	>	<	<	>	<	
s	>	>	>	>	>	>	>	>	>	>	>		>			>		>
$	<	<	<	<	<	<	<	<	<	<	<	<		<	<		<	

Tabulka 4: Precedenční tabulka

Buňka obsahující [chyba / <] je speciálním případem, kdy '<' je pro zpracování argumentů funkce, ale jedná se o chybu, pokud se vyskytne mimo volání funkce.

5.5 Redukční pravidla

E	+ E	→	E	S	<> S	→	B
E	- E	→	E	S	= S	→	B
E	* E	→	E	-E		→	E
E	/ E	→	E	i		→	E ∨ S
E	\ E	→	E	(E)		→	E
E	> E	→	B	(S)		→	S
E	>= E	→	B	(B)		→	B
E	< E	→	B	f()		→	E ∨ S
E	<= E	→	B	f(E)		→	E ∨ S
E	<> E	→	B	f(S)		→	E ∨ S
E	= E	→	B	s		→	S
S	> S	→	B	S	+ S	→	S
S	>= S	→	B	E,E		→	E
S	< S	→	B	E,S		→	E
S	<= S	→	B	S,E		→	E
				S,S		→	E

Tabulka 5: Redukční pravidla

5.6 Metriky kódu

Počet hlavičkových souborů: 11

Počet zdrojových souborů: 11

Počet řádků zdrojového textu: 6357 řádků

Velikost statických dat: 183.4 kB

Velikost spustitelného souboru:
79.1 kB (systém Ubuntu 16.04, 64-bitová architektura, bez ladících informací)
134 kB (systém Windows 10, 64-bitová architektura, bez ladících informací)