## Image result for fit vut logo

## DOKUMENTACE PROJEKTU IFJ a IAL

**Implementace překladače imperativního jazyka IFJ19**

**Tým 65, varianta I**

*Diviš Jan xdivis12 25 %*

*Kopáček Jiří xkopac06 vedoucí**25 %*

*Pojsl Jakub xpojsl00 25 %*

*Sasín Jonáš xsasin05 25 %*

8. 12. 2018

1. **Úvod/zadání**

Cíl projektu bylo vytvořit překladač, který přeloží jazyk ifj19 (zjednodušená verze jazyka Python) do mezikódu ifjcode19, pro který už byl dodán interpret. Program tedy načítá na vstupu kód v jazyce ifj19 a na výstup tiskne instrukce v jazyce ifjcode19.

1. **Práce v týmu**
   1. Rozdělení

Jan Diviš

* Abstraktní datové typy, syntaktická analýza výrazů, dokumentace

Jiří Kopáček

* Vedení projektu, lexikální analýza, abstraktní datové typy, tabulka symbolů, generování vestavěných funkcí, dokumentace

Jakub Pojsl

* Abstraktní datové typy, návrh gramatiky, syntaktická a sémantická analýza výrazů, generování výrazů, dokumentace

Jonáš Sasín

* Návrh gramatiky, syntaktická a sémantická analýza kódu (kromě výrazů), generování kódu (kromě výrazů), dokumentace
  1. Komunikace

Snažili jsme se rozdělit práci tak, aby každý mohl z co největší části pracovat samostatně. Díky tomu nám stačilo minimum osobních setkání a většina komunikace probíhala elektronicky na našem vlastním Discord serveru. Většinou bylo potřeba se pouze domluvit na rozhraní některých funkcí, popřípadě se skupinově zamyslet nad určitým problémem (např. gramatika).

* 1. Správa kódu

Pro správu kódu jsme využívali Git, zdrojový kód jsme sdíleli přes GitHub.

1. **Implementace**

Projekt se skládá z několika hlavních částí:

* 1. Lexikální analýza

Celý lexikální analyzátor je implementován v souboru scanner.c a využívá pomocné datové struktury zásobník a dynamický řetězec. Scanner je implementován standardně podle předem navrženého konečného automatu (jehož diagram je přiložen na konci této dokumentace).

Mezi hlavní části scanneru patří struktura Token, která má 2 složky. První je typ tokenType, který slouží pro identifikaci typu tokenu a druhá je typ tString, což je dynamický řetězec sloužící pro ukládání atributu tokenu (návrh pro tuto implementaci byl inspirován staršími záznamy democvičení k předmětu IFJ).  
Další hlavní částí je funkce get\_next\_token. Této funkci je při volání předán odkaz na nějaký výše zmíněný token a scanner následně přepíše složky tohoto tokenu podle přečtených informací na vstupu. Samotná funkce je převážně tvořena cyklem while, který čte znaky ze vstupu a v něm vnořeném příkazu switch, který přepíná mezi stavy konečného automatu.

**Indentace**

Asi největším problémem při implementaci scanneru byla detekce indentace. Tento problém vedl k vytvoření další části scanneru, a to indentačního zásobníku. Samotný zásobník ukládá typ integer a má jednoduchou strukturu (shodnou se způsobem implementace vyučovaným v předmětu IAL). Scanner si pamatuje, zdali byl poslední token konec řádku. Pokud ano, tak při získávání dalšího tokenu přejde do stavu počítání indentace. V něm počítá všechny mezery až do prvního jiného znaku na řádku. Pokud je tento první znak řádkový komentář nebo konec řádku, je počet mezer zahozen. Pokud ne, je porovnán počet mezer s hodnotou na zásobníku.

Větší hodnota než je na zásobníku vede k pushnutí počtu mezer na zásobník a generování tokenu indent. Stejná hodnota jen zahodí současný počet mezer. Nižší hodnota znamená, že došlo k dedentaci. To vede kromě generování tokenu dedent také k nastavení dedentační fáze scanneru. Ten existuje kvůli případu, kdy je potřeba generovat více dedentací zasebou. Při dalším volání get\_next\_token se před čtením dalších znaků nejdříve zkontroluje, zdali jsme v dedentační fázi. Pokud ano, tak znovu porovnáme dříve získaný počet mezer a hodnotu na vrcholu zásobníku. To budeme opakovat tak dlouho, dokud hledaný počet mezer nenalezneme na vrcholu zásobníku a tím vygenerujeme všechny potřebné tokeny dedent. Dedentační fáze je také užitečná při dosažení konce vstupu, kdy opět nastává dedentační fáze před generováním tokenu EOF s tím, že hledáme indentaci 0. Tím dosáhneme vyprázdnění zásobníku a generování všech zbývajících tokenů dedent.

Další funkce scanneru už zahrnují pouze běžné kontroly, jako například zdali je daný znak alfanumerický nebo zdali je daný řetězec rezervované klíčové slovo.

Ve výsledku lze tedy se scannerem pracovat pouze pomocí funkce get\_next\_token po inicializaci indentačního zásobníku.

* 1. Syntaktická analýza
     1. Rekurzivní sestup
     2. Precedenční analýza
  2. Sémantická analýza
  3. Generování kódu

Hlavní tělo generujeme funkcemi ze souboru builtin\_functions.c. Zde jsou obsaženy zabudované funkce jazyka ifj19 a také hlavní funkce generate\_main\_body, která vygeneruje hlavičku a zabudované funkce a tiskne je na výstup.

* + 1. ??
  1. Pomocné datové struktury

Během implementace jednotlivých částí jsme využili několik datových struktur, většinou probíraných v předmětu IAL.

* + 1. Tabulka symbolů

Tabulka slouží pro ukládání uživatelských proměnných a funkcí a je implementována pomocí binárního vyhledávácího stromu v souboru symtable.c. Detaily implementace této struktury byly navrhnuty na základě poznatků z předmětu IAL. Funkce vyhledávání a vkládání uzlu (a smazání s tím, že smazání samostatného uzlu nakonec nebylo využito) jsou implementovány nerekurzivně, protože se jedná o časté operace. Funkce smazání všech uzlů a prohledání všech uzlů kvůli nedefinovaným funkcím jsou implementovány rekurzivně, obě tyto funkce jsou volány pouze na konci práce s tabulkou symbolů.

Samotné uzly obsahují složky klíč, dále proměnnou, která určuje, zdali je v daném uzlu uložena proměnná nebo funkce, odkazy na levý a pravý uzel a také strukturu tSymdata, která obsahuje informace o symbolu. Konkrétně se jedná o počet parametrů funkce a informaci, zdali byla funkce tohoto uzlu definovaná. Tyto informace původně mohly být obsažený přímo v uzlu stromu, při návrhu jsme si ale ještě nebyli jisti, co vše budeme potřebovat znát o symbolu a tak jsme pro něj vytvořili samostatnou strukturu.

* + 1. Zásobník

Jak už bylo zmíněno, zásobník byl využit při počítání indentace v lexikální analýze a je implementován v souboru stack.c. Dále byl implementován také druhý zásobník využitý při analýze výrazů, který má podobný účel jako běžný zásobník, ale prvky v něm uložené také obsahují odkaz na následující prvek a vrchol tohoto zásobníku je vždy první položka. Kvůli tomu už se také trochu jedná o jednosměrně vázaný seznam, ve kterém lze přidávat a odebírat prvky pouze na začátku seznamu.

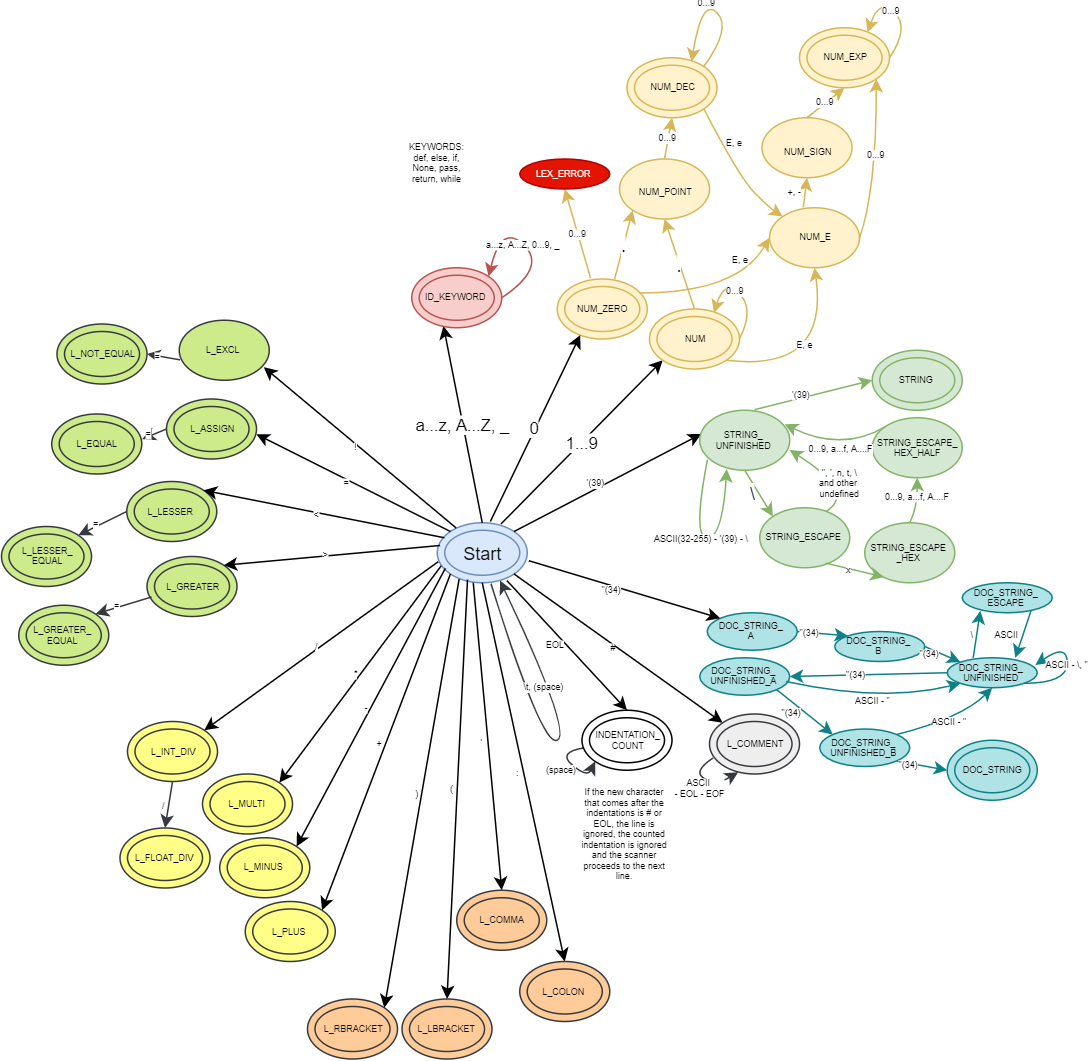
* + 1. Dynamický řetězec

Dynamický řetězec je struktura, která obsahuje odkaz na pole znaků s daným řetězcem a také proměnné délka a alokovaná délka. Tato struktura je využita téměř všude, kde je potřeba skládat nějaký řetězec. Hlavní využití má pro ukládání atributů tokenů, které mohou být nekonečně dlouhé. Při přidávání do řetězce se kontroluje přesáhnutí alokované délky a případně je řetězec realokován.

* + 1. Seznam tokenů

1. **Závěr**

Tým na projekt jsme měli sestavený už před začátkem semestru. Kromě toho už někteří z nás spolu dříve na týmovém projektu pracovali (v rámci předmětu IVS), tudíž jsme s týmovým vývojem už měli zkušenosti a spolupráce nepředstavovala žádný problém. Trochu se nám nepodařilo odhadnout časovou náročnost některých potřebných úkonů, a proto jsme nestihli mít na pokusné odevzdání celý projekt hotový (což jsme původně plánovali) kvůli absenci generování kódu. Tuto část jsme však dělali společně, a tak nebyl problém ji stihnout do finálního odevzdání. Přestože byl projekt poměrně časově náročný, všichni v týmu by asi souhlasili s tvrzením, že se jednalo o jednu z nejzajímavějších a nejzábavnějších částí informatiky, se kterou jsme se doposud na této fakultě setkali. Kromě toho bylo IFJ zatím ten největší projekt, na kterém každý z nás pracoval, což samo o sobě přinese spoustu užitečných zkušeností.

1. **Diagram konečného automatu**
2. **LL-gramatika**

****

1. **LL-tabulka**
2. **Precedenční tabulka**