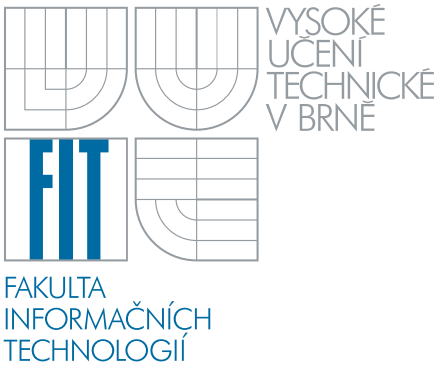



Dokumentace k projektu IFJ/IAL

# Implementace interpretu imperativního jazyka IFJ11

9.prosince 2011

Tým číslo 19, varianta a/2/I
Radka Škvařilová xskvar06 20%
Michal Uhliarik xuhlia01 20%
Kateřina Zaklová xzaklo00 20%
Tomáš Zaoral xzaora02 20%
Libor Zapletal xzaple29 20%
Rozšíření: LENGTH, MODULO, REPEAT

# Obsah

# 1 Úvod

Dokumentace popisuje implementaci překladače imperativního jazyka IFJ11. Program načítá zdrojový soubor, vyhodnotí, zda je kód v pořádku a pokud ano, pak kód interpretuje a vrátí 0. V případě, že najde chybu, vrací kód chyby.

# 2 Zadání

Jazyk IFJ11 je podmnožinou programovacího jazyka Lua, ve kterém vznikají například hry. Jazyk je case sensitive (záleží na velikosti písmen) a je jazykem netypovaným(datový typ každé proměnné určí hodnota do ní vložená).

Program se skládá ze 3 hlavních částí - nejdůležitější části je **syntaktický analyzátor**. Načítá zdrojový soubor prostřednictvím **lexikálního analyzátoru** a překládá kód na posloupnost instrukcí, které následně předá **interpretu** k jejich vykonání.

Pozn. při řešení jsme využili modul str, dostupný z ukázkového projektu.

## 2.1 Varianta zadání

Náš tým řešil variantu projektu a/2/I:

- Knuth-Morris-Prattův vyhledávací algoritmus
- řazení metodou heap sort
- implementace tabulky symbolů binárním vyhledávacím stromem

# 3 Implementace

V této kapitole je popsána implementace jednotlivých částí programu.

## 3.1 Lexikální analýza

Lexikální analyzátor načítá ze zdrojového souboru posloupnosti znaků (lexémy) a zpracovává je pomocí konečného automatu. Komentáře a bílé znaky se zpracují také, ale scanner je nikam neuloží a pokračuje ve zpracování dalšího tokenu. Zpracované lexémy ( = tokeny) jsou předávány syntaktickému analyzátoru. Načítání a zpracování lexémů obstarává funkce `getNextToken()`. Scanner má také funkci zotavení z chyby, v případě, že narazí na chybu, pokusí se dočíst vstup po první adekvátní znak, kterým může začít následující token. Zároveň vrátí typ chybného tokenu (ve zvláštním případě vrací typ EOF).

Tokeny jsou prvky zdrojového souboru, například identifikátory, číselné literály, řetězcové literály aj.

V příloze č. 1 je přiložen model konečného automatu, podle kterého byl naimplementován lexikální analyzátor.

## 3.2 Syntaktická analýza

Syntaktický analyzátor je jádrem překladače. Překládá zdrojový kód na pseudoinstrukce a spouští jejich interpretaci. Pokud nastane chyba, interpretace se nespustí a vypíše se kód chyby.

Na vstupu analyzátor očekává zdrojový kód. Spustí si lexikální analyzátor a pomocí funkce `getNextToken()` načítá posloupnost tokenů. Pro ověření syntaxe načtených tokenů je implementován rekurzivní sestup podle pravidel precedenční a prediktivní LL gramatiky. Výrazy se zpracovávají pomocí precedenční gramatiky metodou zdola nahoru, vše ostatní prediktivní gramatikou metodou shora dolů. Instrukční list se generuje voláním funkce `MakeInst()`.

V příloze č.2 jsou přiložena použitá pravidla LL gramatiky, v příloze č. 3 tabulka vytvořená z těchto pravidel.

### 3.3 Interpret

Interpret provádí interpretaci tříadresného kódu, který generuje syntaktický analyzátor. Tříadresný kód je uložen v instrukčním listu, který je implementován pomocí abstraktního datového typu jednosměrně vázaný lineární seznam.

U aritmeticko-relačních operací provádí interpret sémantickou kontrolu, například typování operandů, přetypování aj.

### 3.4 Binární vyhledávací strom

Tabulka symbolů byla implementována prostřednictvím binárního vyhledávacího stromu, který uchovává identifikátory funkcí a jejich proměnných, jejichž identifikátory jsou uloženy v dalších binárních vyhledávacíh stromech.

Každá funkce zde má svůj vlastní binární vyhledávací strom proměnných. Jako vyhledávací klíč slouží daný identifikátor.

S tabulkou symbolů se pracuje pomocí funkce `tableHandler()` z modulu binarytree.c.

### 3.5 Knuth-Morris-Prattův vyhledávací algoritmus

Algoritmus ke své práci využívá konečný automat, ve kterém z každého uzlu vychází tolik hran, kolik je znaků abecedy. Automat načítá znaky, a pokud dojde ke shodě, posune se na další stav, v opačném případě se vrátí na stav předchozí. Složitost algoritmu je maximálně O(m+n).

Před voláním funkce je ošetřen případ, kdy vyhledávaný podřetězec je prázdný, funkce pak vrátí 0. Pokud je výsledek samotné vyhledávací funkce záporné číslo, interpret hodnotu přetypuje na false.

### 3.6 Heap sort

Heap sort je řadící metoda s linearitmickou složitostí O(N log N). Patří mezi nejlepší řadící metody, ale není stabilní. Hlavní myšlenkou tohoto algoritmu je datová struktura nazývaná halda, pomocí které lze seřadit dodaná data pouhým vložením a následným postupným vybíráním.

Pro větší přehlednost řetězec nejdříve převedeme na pole a následně pole řadíme. Před samotným voláním řadící funkce ošetříme výjimečné stavy (kontrola datového typu, neprázdný řetězec aj.). Řadící funkce zadaný řetězec přepíše, proto si před jejím voláním uložíme ukazatel na původní řetězec.

## 4 Rozšíření

Byla implementována celkem 3 rozšíření:

- length – realizuje výpočet délky řetězce prostřednictvím unárního operátoru #
- modulo – pomocí bináního operátoru % realizuje funkci modulo
- repeat – interpret podporuje cyklus `repeat...until`

## 5 Testování

Od počátku implementace projektu byl stanoven člen týmu, který se staral výhradně o testování. Ostatní členové ho informovali o fázi vypracování svých úkolů, aby mohl současně vytvářet testy k jednotlivým částem. Nakonec vytvořil komplexní testy pro celý projekt v bashi, abychom mohli rychle a efektivně testovat celý program.

## 6 Práce v týmu

Na začátku nám vedoucí rozdělila úkoly, poté jsme se každý týden scházeli ke konzultacím a diskuzi o řešení jednotlivých částí. Dále jsme poměrně dost využívali komunikační nástroje jako instant messenger, email a především naše soukromé diskuzní fórum.

Ke sdílení kódu jsme využili repozitář GIT.

## 7 Závěr

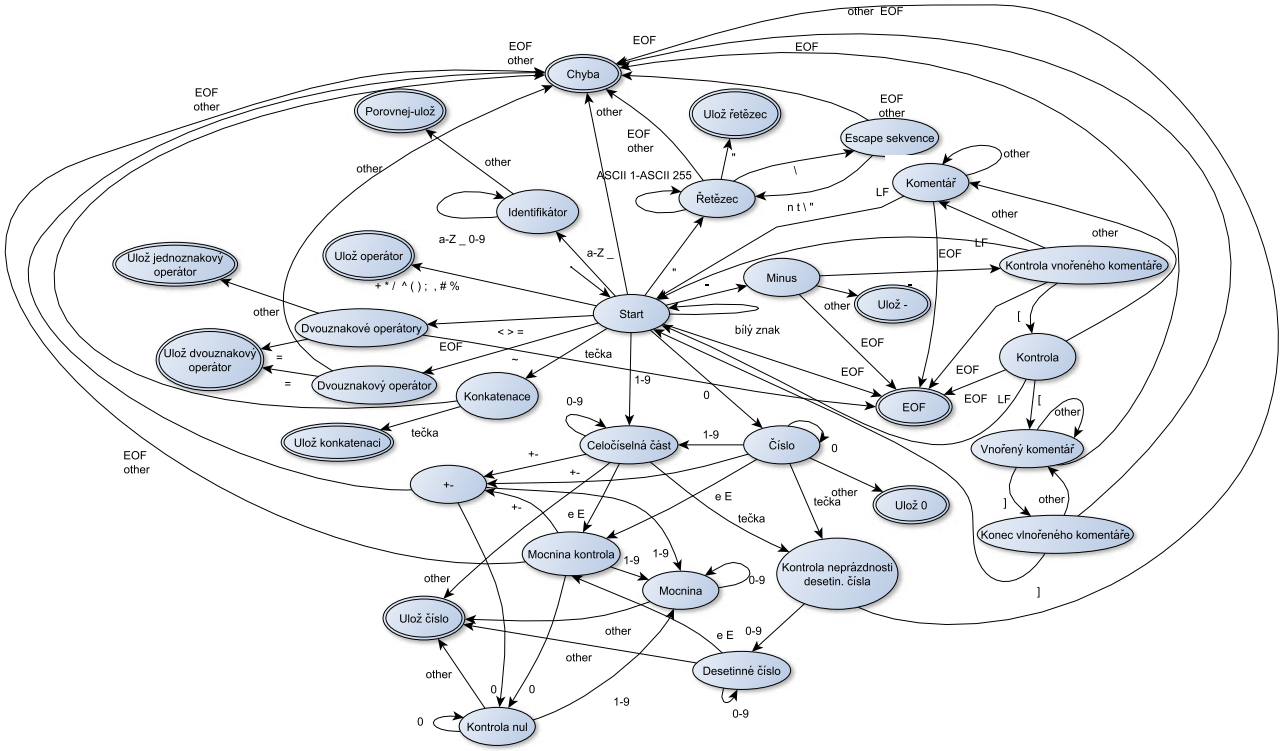
Projekt IFJ11 se nám podařilo víceméně úspěšně naimplementovat i s několika rozšířeními. Odnesli jsme si spoustu zkušeností s prací v týmu. Nejdůležitějším bodem byla bezesporu komunikace mezi členy, tu jsme však důsledně dodržovali. Interpret pracuje dle zadání, byl řádně otestován a dodržuje požadavky zadání na vstupní a výstupní formát dat.

## 8 Metriky kódu

Počet řádků kódu: 6635
Počet zdrojových souborů: 18 + Makefile
Velikost spustitelného souboru: 81750 bytů (Linux 32bit)

# 9 Přílohy

## 9.1 Konečný automat




Obrázek 1: Konečný automat

## 9.2 Pravidla LL gramatiky

1. <program> → <func def list>
2. <func def list> → <func def ><func def_n >;
3. <func def_n> → <func def><func def_n>
4. <func def_n> → $\varepsilon$
5. <func def> → function fid(<argument list>)<var def list><statement list>
6. <argument list> → vid<argument_n>
7. <argument list> → $\varepsilon$
8. <argument_n> → ,vid<argument_n>
9. <argument_n> → $\varepsilon$
10. <var def list> → <var def>;<var def list>
11. <var def list> → $\varepsilon$
12. <var def> → local vid<initialization>
13. <initialization> → =<factor>
14. <initialization> → $\varepsilon$
15. <statement list> → <statement>;<statement list>
16. <statement list> → $\varepsilon$
17. <statement> → vid=<assignment>
18. <statement> → write(<expression list>)
19. <statement> → return<expression>
20. <statement> → if<expression>then<statement list>else<statement list>end
21. <statement> → while<expression>do<statement list>end
22. <statement> → repeat<statement list>until<expression>
23. <assignment> → read(<input>)
24. <assignment> → fid(<param list>)
25. <assignment> → <expression>
26. <expression list> → <expression><expression_n>
27. <expression list> → $\varepsilon$
28. <expression_n> → ,<expression><expression_n>
29. <expression_n> → $\varepsilon$
30. <input> → string
31. <input> → integer
32. <input> → real
33. <param list> → <param><param_n>
34. <param list> → $\varepsilon$
35. <param_n> → ,<param><param_n>
36. <param_n> → $\varepsilon$
37. <param> → <factor>
38. <param> → vid
39. <factor> → string
40. <factor> → integer
41. <factor> → real
42. <factor> → boolean
43. <factor> → nil

### 9.3 Precedenční tabulka

| | == | ∼= | <= | >= | < | > | .. | + | - | * | / | % | # | -(un) | ^ | ( | ) | vid | DT | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| == | > | > | > | > | > | > | < | < | < | < | < | < | < | < | < | < | > | < | < | > |
| ∼= > | > | > | > | > | > | > | < | < | < | < | < | < | < | < | < | < | > | < | < | > |
| <= | > | > | > | > | > | > | < | < | < | < | < | < | < | < | < | < | > | < | < | > |
| >= | > | > | > | > | > | > | < | < | < | < | < | < | < | < | < | < | > | < | < | > |
| < | > | > | > | > | > | > | < | < | < | < | < | < | < | < | < | < | > | < | < | > |
| > | > | > | > | > | > | > | < | < | < | < | < | < | < | < | < | < | > | < | < | > |
| .. | > | > | > | > | > | > | > | < | < | < | < | < | < | < | < | < | > | < | < | > |
| + | > | > | > | > | > | > | > | > | > | < | < | < | < | < | < | < | > | < | < | > |
| - | > | > | > | > | > | > | > | > | > | < | < | < | < | < | < | < | > | < | < | > |
| * | > | > | > | > | > | > | > | > | > | > | > | > | < | < | < | < | > | < | < | > |
| / | > | > | > | > | > | > | > | > | > | > | > | > | < | < | < | < | > | < | < | > |
| % | > | > | > | > | > | > | > | > | > | > | > | > | < | < | < | < | > | < | < | > |
| # | > | > | > | > | > | > | > | > | > | > | > | > | > | > | < | < | > | < | < | > |
| -(un) | > | > | > | > | > | > | > | > | > | > | > | > | > | > | < | < | > | < | < | > |
| ^ | > | > | > | > | > | > | > | > | > | > | > | > | > | > | < | < | > | < | < | > |
| ( | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | = | < | < | |
| ) | > | > | > | > | > | > | > | > | > | > | > | > | | | > | | > | | | > |
| vid | > | > | > | > | > | > | > | > | > | > | > | > | | | > | | > | | | > |
| DT | > | > | > | > | > | > | > | > | > | > | > | > | | | > | | > | | | > |
| $ | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | < | | < | < | |