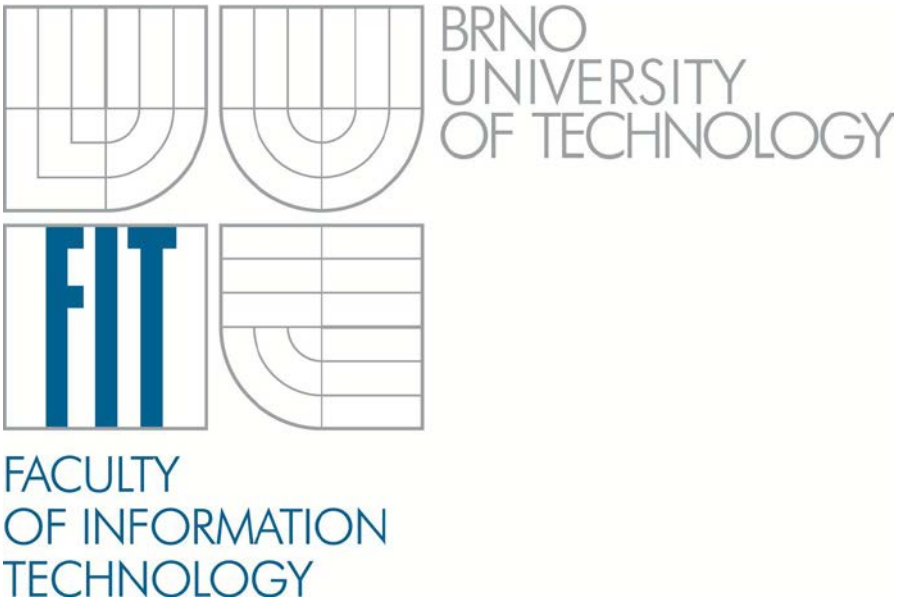



Dokumentace k projektu pro předměty IAL a IFJ

# Implementace interpretu imperativního jazyka IFJ10

5. prosince 2010


| | | |
|---|---|---|
| Skupina: | 72 | |
| Varianta: | a/4/I | |
| Řešitelé: | 20% - Martin Knapovský | xknapo02@stud.fit.vutbr.cz |
| | 20% - Jan Myler | xmyler00@stud.fit.vutbr.cz |
| | 20% - Pavel Antolík | xantol00@stud.fit.vutbr.cz |
| | 20% - Jiří Navrátil | xnavra36@stud.fit.vutbr.cz |
| | 20% - Ondřej Kratochvíl | xkrato05@stud.fit.vutbr.cz |


Rozšíření: Zpracování čísel se základem 2, 8 a 16

## Úvod
Tato dokumentace je součástí projektu pro předměty IAL a IFJ, jehož náplní bylo implementovat interpret imperativního jazyka IFJ10.

## Varianta zadání
Zvolili jsme variantu a/4/I s následujícím významem:
**a** – implementace vestavěné funkce find pomocí Knuth-Moris-Prattova algoritmu
**4** – implementace vestavěné funkce sort pomocí algoritmu Merge sort
**I** – implementace tabulky symbolů pomocí binárního vyhledávacího stromu

## Rozdělení interpretu na logické části
**Lexikální analyzátor** čte vstupní soubor po znacích a reprezentuje ho pomocí tokenů.
**Syntaktický analyzátor** přijímá tokeny lexikálního analyzátoru a pomocí nich kontroluje syntaktickou správnost. Je rozdělen na syntaktickou analýzu „shora-dolů“, která řídí celou interpretaci a syntaktickou analýzu „zdola-nahoru“ používaná pro kontrolu výrazů.
**Sémantický analyzátor** kontroluje v průběhu syntaktické analýzy sémantickou správnost programu, který interpretujeme. Mezi tyto kontroly patří například deklarovanost proměnných, nebo datové typy použité ve výrazech.
**Interpretační modul** zpracovává výstup ostatních částí a vytváří cílový kód.

## Implementační rozčlenění
Program je rozdělen do modulů, což umožňuje lepší orientaci v kódu programu a jeho snadnější správu.

| | |
|---|---|
| **const** | - modul, který definuje chybové kódy interpretu |
| **dllist** | - implementace seznamu pro interpretaci |
| **ilist** | - modul instrukčního seznamu |
| **ial** | - vestavěné funkce a tabulka symbolů |
| **interpret** | - modul interpretu |
| **main** | - řídící modul |
| **parser** | - implementace syntaktické analýzy „shora-dolů“ metodou rekurzivního sestupu a také precedenční syntaktické analýzy „zdola-nahoru“ pro zpracování výrazů. |
| **prec_stack** | - zásobník precedenční analýzy |
| **scanner** | - konečný automat lexikálního analyzátoru |
| **str** | - knihovna pro práci s potencionálně nekonečnými řetězci |

## Návratové hodnoty interpretu
**0** – Vše v pořádku
**1** – Chyba lexikální struktury vstupního souboru
**2** – Chyba syntaktické struktury vstupního souboru
**3** – Chyba vstupního souboru v rámci sémantických kontrol
**4** – Chyba za běhu programu v rámci interpretace konkrétních dat
**5** – Interní chyba interpretu neovlivněna vstupním programem

# Detailní popis jednotlivých částí

## Lexikální analyzátor

Lexikální analyzátor je implementován jako konečný automat, jehož diagram je uveden v příloze. Funkce *int get_next_token(string* attr)*, jakožto samotný automat a hlavní součást modulu, prochází zdrojový soubor po znacích a na základě čteného znaku rozhoduje o své činnosti. Může tak přejít do nekoncového stavu a čekat na další znak, který by mu umožnil přechod do jiného nekoncového stavu, nebo do stavu koncového. Při přijetí znaku, které neodpovídá rozpracovanému lexému, resp. pokud automat nemůže s konečným počtem přečtených znaků přejít do koncového stavu, je volajícímu vrácen kód pro chybu lexikální struktury vstupního souboru a v atributu vráceno číslo řádku, kde se chyba nachází. Pokud čtené znaky umožňují přechod do koncového stavu, znamená to, že byl rozpoznán lexém a je vrácen tzv. *token*, který reprezentuje typ rozpoznaného lexému a jeho atributy.

### Klíčová slova

Jazyk IFJ10 obsahuje klíčová slova, která musí být odlišena od identifikátorů, které definuje uživatel. Rozpoznání klíčového slova zprostředkovává funkce *int is_keyword(string* attr)* porovnávající námi přečtený řetězec ze zdrojového souboru s řetězci uloženými v tabulce *keywords*, kde jsou klíčová slova definována. Při nalezení řetězce v tabulce je vrácena odpovídající návratová hodnota, která definuje, o které klíčové slovo se konkrétně jedná.

Tabulka keywords:

```
const char* keywords[TABLE_SIZE] = {
"AS"          ,"DECLARE"    ,"DIM",
"DO"          ,"DOUBLE"     ,"ELSE",
"END"         ,"FIND"       ,"FUNCTION",
"IF"          ,"INPUT"      ,"INTEGER",
"LOOP"        ,"PRINT"      ,"RETURN",
"SCOPE"       ,"SORT"       ,"STRING",
"THEN"        ,"WHILE",
};
```

### Příklad funkce lexikálního analyzáru

Čteme řetězec „*hi “*. Lexikální analyzátor přečte znak *h* a z počátečního stavu *S_START* přejde do stavu *T_ID* a znak *h* přidá do atributu, který byl předtím prázdný. Analyzátor čte další znak, avšak tentokrát již pokračuje od stavu *T_ID*. Se znakem *i* přechází opět do stavu *T_ID* a přidá znak k atributu. Nyní je přečtena mezera, kterou lexikální analyzátor vrátí zpět do souboru, jelikož se již nachází v koncovém stavu a s mezerou z něj nemůže přejít jinam. Pomocí funkce *is_keyword( )* zjistí, že se nejedná o klíčové slovo a vrátí token typu *T_ID* a atributem *„hi“*.

## Syntaktický analyzátor

Syntaktický analyzátor je jednou z hlavních částí tohoto projektu. Obecně má za úkol kontrolovat, zda řetězec tokenů vstupního zdrojového kódu reprezentuje správně napsaný program v jazyce IFJ10.

Správnost syntaxe překládaného či interpretovaného programu je dána nalezením tzv. derivačního stromu, jehož vytváření je založeno na gramatických pravidlech zvoleného programovacího jazyka.

### Syntaktický analyzátor „shora-dolů“

Nejprve bylo nutné analyzovat zadání projektu a vytvořit gramatická pravidla pro jazyk IFJ10. Na základě těchto pravidel byla zkonstruována bezkontextová gramatika s epsilon pravidly (viz. příloha), ze které vychází LL(1) tabulka, na základě které je implementována syntaktická analýza „shora-dolů“ metodou rekurzivního sestupu.

Pro každý neterminál LL tabulky je vytvořena odpovídající funkce, jenž simuluje provádění gramatických pravidel. V rámci pravidel gramatiky rozlišujeme terminály a neterminály. Terminály jsou konečné řetězce nad vstupní abecedou odpovídající pravidlům gramatiky jazyka IFJ10. Neterminály jsou pomocné symboly, které je třeba pomocí dalších pravidel v několika derivačních krocích nahradit sekvencí terminálů.

Simulace pravidel gramatiky je prováděna následujícím způsobem:

- požádej lexikální analyzátor o nový token,
- jestliže je očekáván terminál, porovnej typ načteného tokenu s očekávaným,
- pokud typy neodpovídají – syntaktická chyba,
- jestliže je očekáván neterminál, zavolej funkci, která simuluje pravidlo gramatiky odpovídající očekávanému neterminálu.

Takto se pomocí vzájemného volání funkcí a samozřejmě s využitím rekurze simuluje tvorba derivačního stromu.

### Syntaktický analyzátor „zdola-nahoru“

Syntaktický analyzátor „zdola-nahoru“ využívá precedenční analýzy. Ta je implementována pomocí automatu s pěti stavy, kde dva z nich zde nejsou uvedeny, jelikož jsou pouze pro rozlišení chybového výstupu.

Tři hlavní stavy :

- `"="` – přidání tokenu na zásobník
- `"<"` – označení začátku pravé strany gramatiky a přidání tokenu na zásobník
- `">"` – redukce pravé strany gramatiky na levou, vyvolání sémantických akcí a vytvoření instrukcí

Rozhodování, který stav nastane, je prováděno za pomoci precedenční tabulky (viz. příloha), kde se jako index řádku použije převedená hodnota vrchního terminálu v zásobníku a jako index sloupce převedená hodnota vstupního tokenu.

Redukce je provedena za pomoci gramatik pro precedenční analýzu (viz. příloha). Při redukci se dále provádějí sémantické akce, podle kterých jsou vytvářeny instrukce, které jsou ukládány do seznamu instrukcí.

## Knihovna pro práci s řetězci

Jako základ modulu byla použita implementace z ukázkového projektu, která byla doplněna o funkci pro konkatenaci řetězců a upravena tak, aby odpovídala námi dohodnutému stylu formátování kódu a komentářů.

## Vestavěná funkce find

Pro implementaci vestavěné funkce *int find(string *haystack, string *needle)* byl zvolen Knutt-Morris-Prattův algoritmus. Tento algoritmus se liší od ostatních vyhledávacích algoritmů tím, že nejdříve prohledá vyhledávané slovo a na základě opakování jeho prefixu sestaví tzv. *fail-vektor*, jehož počet prvků je roven počtu znaků vyhledávaného slova. Poté, co je tento vektor sestaven, začne samotné vyhledávání podřetězce. *Fail-vektor* je v podstatě jednoduchý stavový automat (implementovány ve formě pole), který říká samotnému vyhledávacímu algoritmu, kde má ve zdrojovém textu začít znovu vyhledávat při nalezení první neshody.

### Ukázka Knutt-Morris-Prattova algroritmu

Pro lepší ilustraci uvedeme příklad takového vektoru pro slovo *„testech“*.

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | t | e | s | t | e | c | h |
| | -1 | 0 | 0 | 0 | 1 | 2 | 0 |

Na první pohled lze vidět, že ve vyhledávaném slově se nám opakuje řetězec *„te“*. V případě, že by došlo k neshodě na šestém znaku (*„c“*), se pomocí údaje na příslušné pozici *fail-vektoru* přeskočí požadovaný počet znaků v prohledávaném textu a nastaví se index ve vyhledávaném slově tak, aby nedocházelo k žádnému zbytečnému vyhledávání. Hodnota *-1* na nulté pozici *fail-vektoru* plní funkci jakési zarážky. Příklad užití *fail-vektoru*:

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 12 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H[i] | t | e | s | t | e | g | h | n | a | t | e | x | t | e | c | h |
| n[i] | t | e | s | t | e | c | h | | | | | | | | | |

Hodnoty proměnných při začátku porovnávání šestého znaku: m = 0, i = 5

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 12 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H[i] | t | e | s | t | e | g | h | n | a | t | e | x | t | e | c | h |
| n[i] | | | | t | e | s | t | e | c | h | | | | | | |

Nová hodnota m: m = m + i – fail[i] => m = 0 + 5 – 2 => m = 3
Nový index: i = fail[i] => i = 2

Vysvětlivky: m - index, který se zvyšuje v případě neshody, v podstatě označuje začátek každého nového vyhledávání
i - index určující, kolikátý znak z vyhledávaného slova právě porovnáváme

V případě prvních pěti znaků proběhne porovnání úspěšně, ale na šestém znaku nastane problém (*„g“ != „c“*). Z hodnot pomocných proměnných a *fail-vektoru* se vypočítá nový počátek vyhledávání (*m = 3*) s tím, že o prvních dvou znacích díky *fail-vektoru* víme, že se opakují, a můžeme začít porovnávat rovnou z třetí pozice vyhledávaného řetězce. Dochází tak k významné úspoře výpočetního času – časová složitost vyhledání je jen Θ(n) (s nutností

předzpracování o složitosti Θ(m)), což je výrazný pokrok oproti ostatním algoritmům se složitostí až Θ((n-m+1)*m). V naší implementaci je funkce členěna na dva while cykly – první obstarává sestavení *fail-vektoru*, ve druhém probíhá samotné vyhledávání.

## Vestavěná funkce sort

Jako algorimus pro implementaci funkce *sort* byl zvolen MergeSort. Podobně jako třeba QuickSort se jedná o stabilní algoritmus typu „rozděl a panuj“. Jeho nevýhodami jsou však o něco nižší rychlost řazení než u zmíněného QuickSortu, nebo např. HeapSortu, a také nutnost použití pomocného pole. Algoritmus pracuje na následujícím principu:

- Pole znaků (v našem případě vstupní řetězec) je rozděleno na dvě části, které mají přibližně stejnou velikost
- Jednotlivé části (podproblémy) jsou za pomocí rekurze seřazeny
- Seřazené podmnožiny postupně spojujeme do jedné výsledné množiny

V naší implementaci funkce *string sort(string *srcstr)* pouze připravuje vstupní a návratová data (např. vytváří ono pomocné pole), k samotnému třídění pak volá funkci *void merge_sort(char input[], char temp[], int left, int right).* V té je nejdříve vstupní pole znaků rozděleno na dvě zhruba stejně velké podmnožiny, na které je potom rekurzivně volána tatáž funkce, dokud se nedosáhne úplného setřídění obou množin, přičemž k jejich spojování je volána funkce *void merge(char input[], char temp[], int left, int mid, int right)*, která dvě setříděné množiny „slije“ zpátky do jednoho pole. Po seřazení celé vstupní množiny se řízení vrátí zpět funkci sort, která překopíruje setříděné pole do struktury string a tu vrátí jako návratovou hodnotu.

## Implementace tabulky symbolů

Pro tabulku symbolů byla v zadání zvolena varianta implementace pomocí binárního vyhledávacího stromu (dále jen BVS). BVS je datová struktura založená na binárním stromu (každý uzel má maximálně dva potomky), pro jehož každý uzel platí, že klíče uzlů v levém podstromu jsou vždy menší než klíč ve vybraném uzlu a klíče v pravém podstromu jsou vždy větší než klíč v uzlu.

Jako základní kostra byly využity zdrojové kódy z domácího úkolu do předmětu IAL. Konkrétně byla použita rekurzivní implementace BVS, která byla upravena do námi požadované podoby, a doplněna dalšími funkcemi, které usnadňují práci s tabulkami symbolů.

### Globální tabulka symbolů

Pro potřeby sémantické analýzy a následné interpretace zdrojového kódu v jazyce IFJ10 byly vytvořeny dva typy tabulky symbolů. První z nich – globální tabulka symbolů (GTS) – obsahuje informace o definicích funkcí.

Pro každou z funkcí je v GTS vytvořen nový uzel, jehož klíčem je identifikátor funkce. Nově vytvořený uzel je navázán na patřičné místo v rámci struktury stromu a do jeho datové části jsou postupně vloženy informace o počtu formálních parametrů funkce, návratovém datovém typu funkce a informace jestli byla funkce pouze deklarována nebo i definována.

### Lokální tabulka symbolů

Pro každou takto vloženou funkci je zároveň vytvořena lokální tabulka symbolů (LTS). Ukazatel na ni je uložen v datové části GTS.

Uzly této tabulky symbolů obsahují informace o formálních parametrech dané funkce a také o jejích lokálních proměnných. Klíčem pro uzly LTS je identifikátor formálního parametru, nebo lokální proměnné. V datové části každého uzlu je uložena informace o datovém typu proměnné a relativním umístění (offsetu) její hodnoty v rámci bloku programového zásobníku.

## Popis instrukční sady a interpretu

Naši instrukční sadu tvoří několik jednoduchých instrukcí deklarovaných v hlavičkovém souboru *ilist.h*. Volání interpretu se provádí po úspěšném skončení sémantické a syntaktické analýzy, jež jako svůj cílový výstup generují posloupnost instrukcí tříadresného kódu. Jednotlivé adresy ukazují do lokální tabulky symbolů. Pro vlastní implementaci interpretu byl vytvořen dvojsměrný lineární seznam, kam se ukládají informace o proměnných funkcí. Pomocí base pointeru se pohybujeme po rámcích daných funkcí.
Ukázka sady instrukcí:

```
//matematické operace
#define I_ADD_I   10    //sčítání integerů
#define I_ADD_D   11    //sčítání doublů
#define I_SUB_I   12    //odčítání integerů
#define I_SUB_D   13    //odčítání doublů
#define I_MUL_I   14    //násobení integerů
#define I_MUL_D   15    //násobení doublů
#define I_IDIV    16    //celošíselné dělení
#define I_DIV     17    //dělení
```

# Doplňkové informace

## Práce v teamu

Implementace projektu byla z počátku semestru naplánována a byla stanovena data dokončení jednotlivých částí. S ohledem na ucelenost zdrojových kódů byla stanovena norma formátování a komentování kódů. Kontrola výsledků práce a řešení nejasností probíhalo na schůzkách, které probíhali s odstupem jednoho týdne. Pro komunikaci byl využit nástroj google wave a pro správu verzí zdrojových souborů a jejich sdílení jsme použili nástroj subversion.

## Metriky projektu

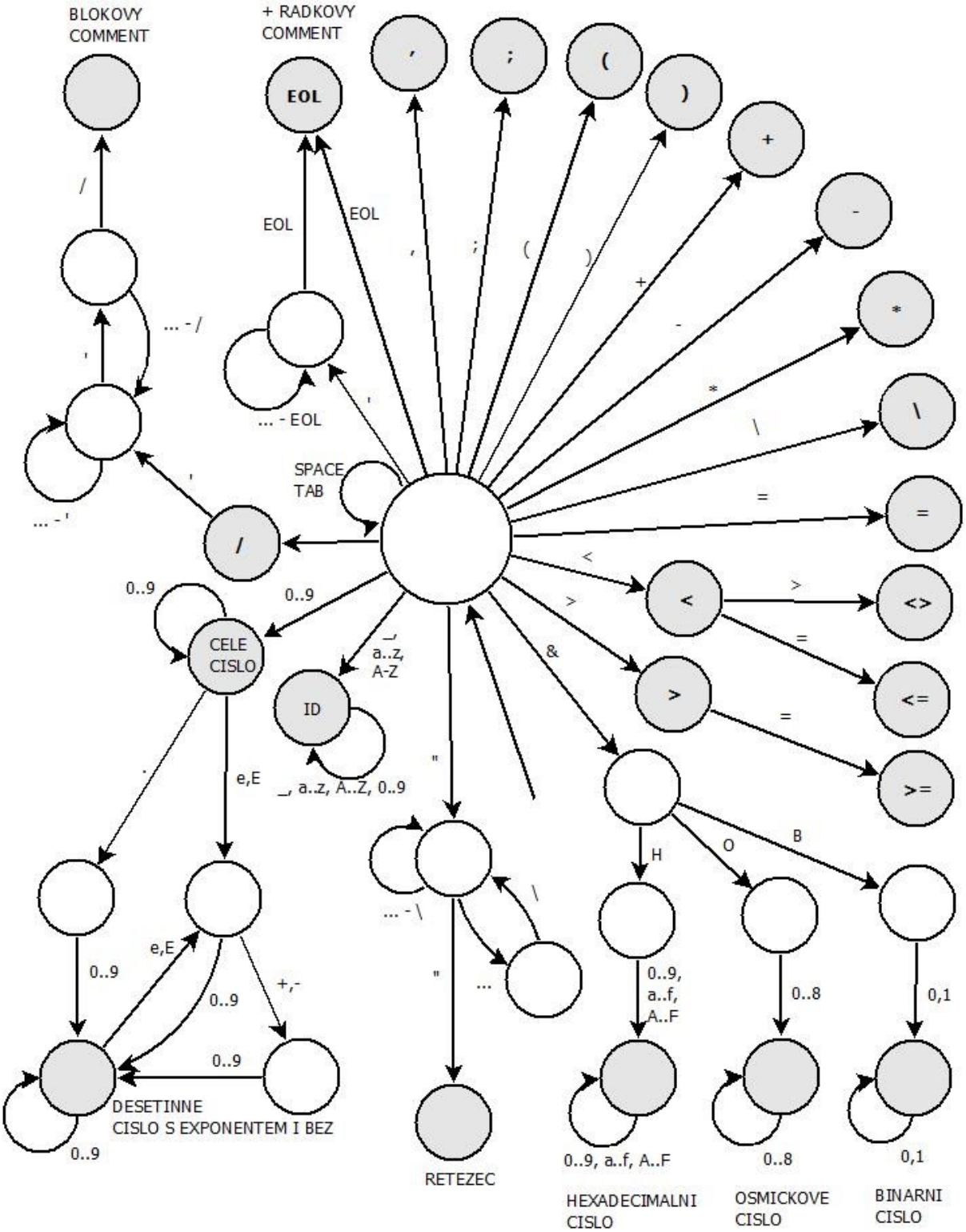
5951 řádků kódu

## Použitá literatura

Zápisky z přednášek IFJ a IAL.

# Přílohy

## Schéma lexikálního analyzátoru



## Gramatika

| | |
|---|---|
| 1 | <prog> → <fn-list> **scope eol** <main-p> **eof** |
| 2 | <fn-list> → <func> **eol** <fn-list> |
| 3 | <fn-list> → **ε** |
| 4 | <func> → **declare function id (** <it-list> **) as** <type> |
| 5 | <func> → **function id (** <it-list> **) as** <type> **eol** <dc-list> <st-list> **end function** |
| 6 | <it-list> → **id as** <type> <it-list2> |
| 7 | <it-list> → **ε** |
| 8 | <it-list2> → **, id as** <type> <it-list2> |
| 9 | <it-list2> → **ε** |
| 10 | <type> → **integer** |
| 11 | <type> → **double** |
| 12 | <type> → **string** |
| 13 | <dc-list> → <decl> **eol** <dc-list> |
| 14 | <dc-list> → **ε** |
| 15 | <decl> → **dim id as** <type> |
| 16 | <st-list> → <stat> **eol** <st-list> |
| 17 | <st-list> → **ε** |
| 18 | <id-list> → **, id** <id-list> |
| 19 | <id-list> → **ε** |
| 20 | <ex-list> → <expr> **;** <ex-list> |
| 21 | <ex-list> → **ε** |
| 22 | <stat> → **input ; id** <id-list> |
| 23 | <stat> → **print** <expr> **;** <ex-list> |
| 24 | <stat> → **id =** <rhs> |
| 25 | <stat> → **if** <expr> **then eol** <st-list> **else eol** <st-list> **end if** |
| 26 | <stat> → **do while** <expr> **eol** <st-list> **loop** |
| 27 | <stat> → **return** <expr> |
| 28 | <main-p> → <dc-list> <st-list> **end scope eol** |
| 29 | <rhs> → **id** <rest> |
| 30 | <rest> → **(** <pm-list> **)** |
| 31 | <pm-list> → <param> <pm-list2> |
| 32 | <pm-list> → **ε** |
| 33 | <pm-list2> → **,** <param> <pm-list2> |
| 34 | <pm-list2> → **ε** |
| 35 | <param> → <type> |

## Precedenční tabulka

```
char pre_table[15][15]={
//        0+  1-  2*  3\  4/  5(  6) 7ID  8= 9<> 10< 11<= 12> 13>= 14$
/* + */ {'>','>','<','<','<','<','>','<','>','>','>','>','>', '>','>','>'},
/* - */ {'>','>','<','<','<','<','>','<','>','>','>','>','>', '>','>','>'},
/* * */ {'>','>','>','>','>','>','<','>','<','>','>','>','>', '>','>','>'},
/* \ */ {'>','>','<','>','<','<','>','<','>','>','>','>','>', '>','>','>'},
/* / */ {'>','>','>','>','>','>','<','>','<','>','>','>','>', '>','>','>'},
/* ( */ {'<','<','<','<','<','<','=','<','<','<','<','<','<', '<','<','-'},
/* ) */ {'>','>','>','>','>','>',' ','>',' ','>','>','>','>', '>','>','>'},
/* ID */{'>','>','>','>','>','>',' ','>',' ','>','>','>','>', '>','>','>'},
/* = */ {'<','<','<','<','<','<','>','<','>','>','>','>','>', '>','>','>'},
/* <> */{'<','<','<','<','<','<','>','<','>','>','>','>','>', '>','>','>'},
/* < */ {'<','<','<','<','<','<','>','<','>','>','>','>','>', '>','>','>'},
/* <= */{'<','<','<','<','<','<','>','<','>','>','>','>','>', '>','>','>'},
/* > */ {'<','<','<','<','<','<','>','<','>','>','>','>','>', '>','>','>'},
/* >= */{'<','<','<','<','<','<','>','<','>','>','>','>','>', '>','>','>'},
/* $ */ {'<','<','<','<','<','<','-','<','<','<','<','<','<', '<','<',' '}
};
```

## Gramatika precedenční analýzy

```
E → E + E
E → E – E
E → E * E
E → E / E
E → E \ E
E → E = E
E → E < E
E → E ≤ E
E → E > E
E → E ≥ E
E → E ≠ E
E → ( E )
E →  VAL
E →  ID
```

## Celkové schéma interpretu