VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií

Implementace interpretu imperativního jazyka IFJ16

Tým 021, varianta a/3/I

Vedoucí: Kyzlink Jiří (xkyzli02)
Kubiš Juraj (xkubis15)
Korček Juraj (xkorce01)
Kubica Jan (xkubic39)
Kovařík Viktor (xkovar77)

10. prosince 2016 Brno

Obsah

1 Úvod

V této dokumentaci naleznete popis návrhu a implementace interpretu jazyka IFJ16, který je velmi zjednodušenou podmnožinou jazyka Java SE 8, což je staticky typovaný objektově orientovaný jazyk. Vybrali jsme si variantu varianta a/3/I, kde jsme měli za úkol přidat do interpretu vestavěnou funkci find, která využívala Knuth-Morris-Prattův algoritmus a funkci sort, kterou jsme měli implementovat tak, aby využívala shell sort.

–bude ještě doplněno-

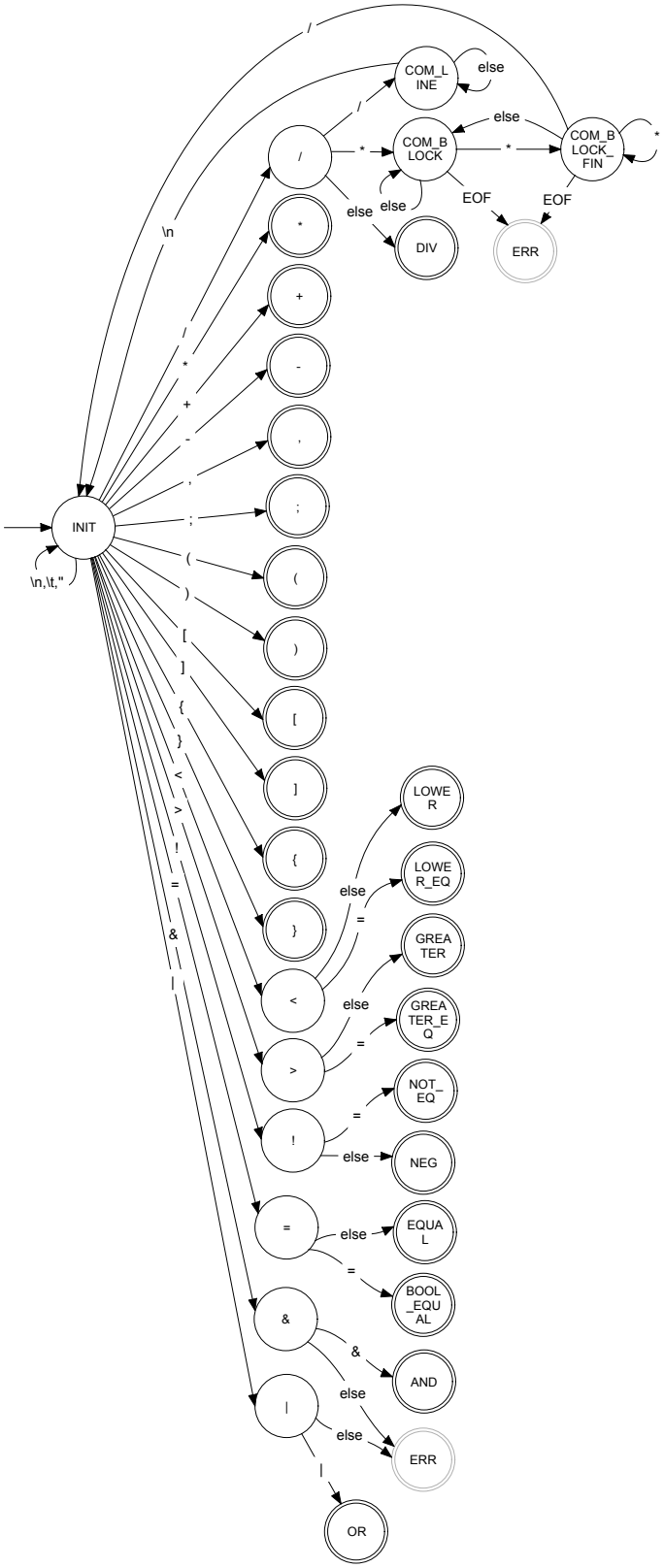
2 Lexikální analyzátor

Lexikální analyza je založena na deterministickém konečném automatu(dále jen *KA*), jehož vstupem je zdrojový kód programu. Lexikální analyzátor na základě předem definovaných pravidel rozdělí jednotlivé posloupnosti znaků na lexémy, které jsou poslány syntaktickému analyzátoru ve formě tokenu. V tokenu jsou obsaženy informace o typu rozpoznaného lexému, jeho délce, pozici v zdrojovém kódu a odpovídájícimu řetězci ze zdrojového kódu. Vedlejším úkolem lexikální analýzy je odstraňování bílých znaků, ale i řádkových a blokových komentářů. Lexikální analýza je řízena syntaktickou analýzou, která postupně žadá o tokeny. Naše implementace obsahuje funkci `peek_token`, která usnadňuje práci syntaktické analýze, protože umožňuje se podívat o jeden token napřed. Klíčové slova, která jse nemůžou vyskytovat v jménech identifikátorů jsou realizovana jako pole řetezců.

Pokud lexikalní analýza narazí na nerozpoznatelný lexém, tak na chybový výstup vypíše hlášení o chybě a řádku na kterém k ní přišlo.

2.1 Diagram konečného automatu lexikální analýzy

KA hlavní schéma:

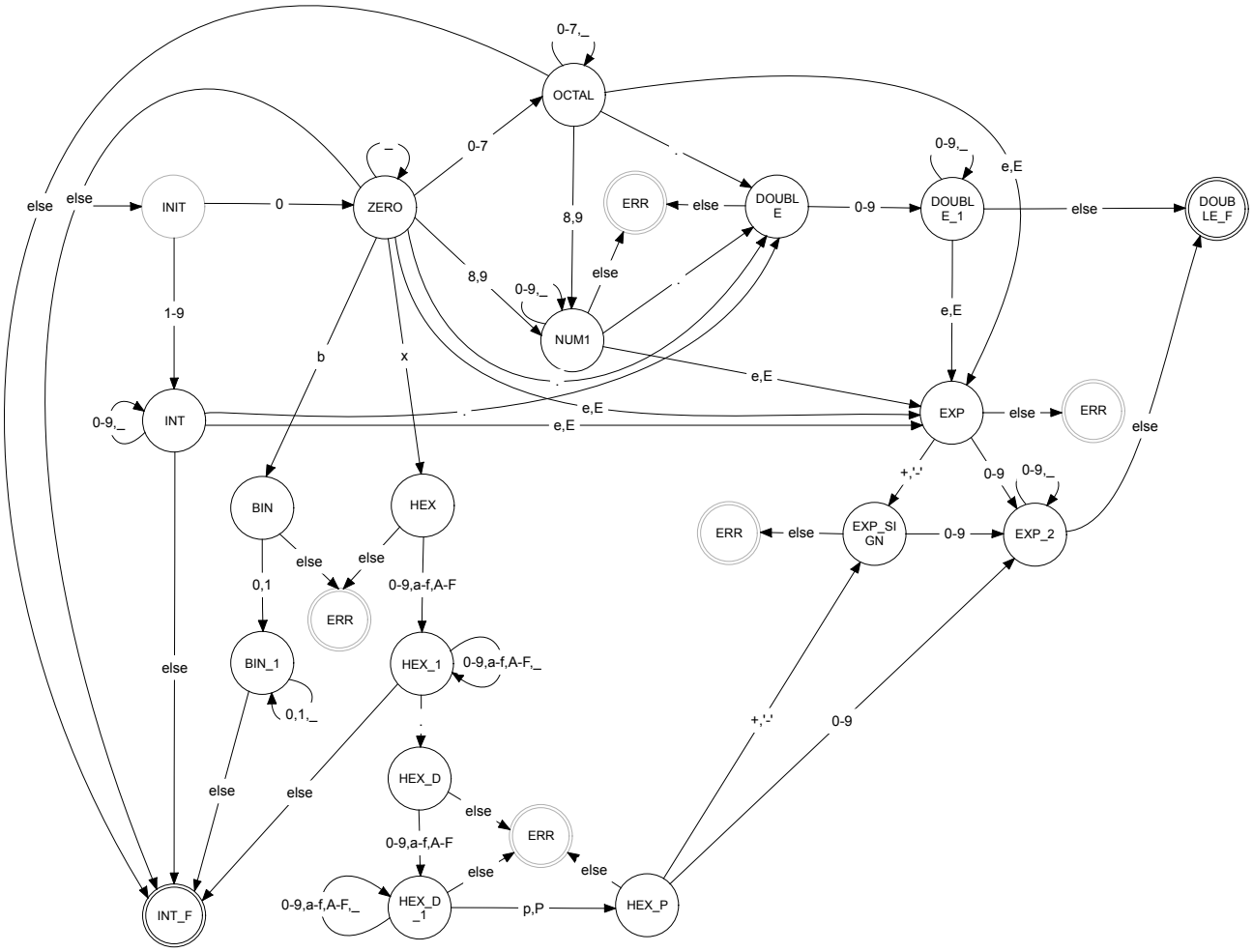


Obrázek 1: KA hlavní schéma

KA chyba:

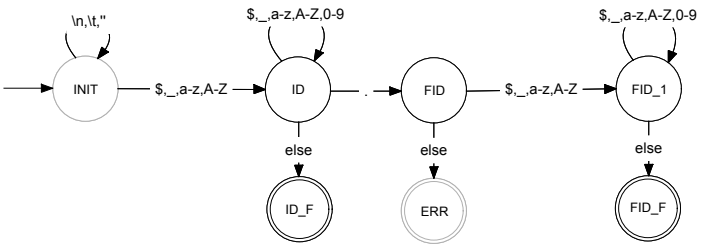
Obrázek 2: KA chyba

KA číslicový literál:



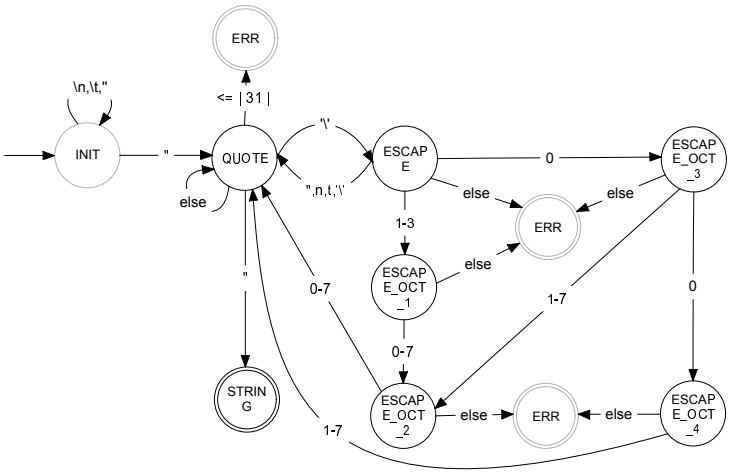
Obrázek 3: KA číslicový literál

KA identifikátor:



Obrázek 4: KA identifikátor

KA řetězec:



Obrázek 5: KA řetězec

3 Syntaktický analyzátor

Syntaktický analyzátor (dále jen *SA*) slouží k vyhodnocování správnosti syntaxe a v našem případě i ke generování abstraktního syntaktického stromu (dále jen *AST*). Vstupen SA je proud tokenů z LA, výstupem je struktura na listingu níže. Struktura obsahuje tybulku globálních symbolů, pro případ kontroly typů, seznam definovaných funkcí, kde každá obsahuje vlastní AST a v posledním prvku je uložen počet globální proměnných pro potřeby alokace paměti.

```
typedef struct {
        Symbo_tree global_symbols;
        Function_list functions;
        int globals;
} Syntax_context;
```

Kód 1: Výstupní struktura SA

3.1 Syntaktická analýza kódu

Syntaktická analýza kódu je implementovaná pomocí rekurzivního sestupu

3.2 Syntaktická analýza výrazů

Úlohou syntaktichého analyzátoru výrazov je správne transformovať výraz na vstupe do podoby AST. Musí pri tom brať na zreteľ prioritu jedtnotlivých operátorov a ich asociativitu.

Zakladom celého analyzátoru je množina pravidiel, precedenčná tabuľka operátorov (ďalej len *tabuľka*) a zásobník. Analýza prebieha spôsobom, že sa prečíta neterminál (token) na vstupe a v tubuľke sa vyhľadá jeho vzťah k neterminálu, ktorý je najbližie vrcholu zásobníka. Tento vzťah určuje, aká akcia je na zásobníku vykonaná (jednotlivé akcie budú vysvetlené nižšie). Následne je prečítaný ďalší token a celý postup sa opakuje do doby, kedy token na vstupe signalizuje koniec výrazu a na zásobnníku sa nenachádza žiadny neterminál.

3.2.1 Precedenčná tabuľka operátorov

Tabuľka vyjadruje vztah medzi každou dvojicou operátorov a určuje aká akcia je na zásobníku vykonaná. Tieto akcie sú:

Symbol	Význam	Sématická akcia
=	Push	Token na vstupe je vložený na zásobník
<		Token na vstupe je vložený na zasobník,
>	Handle	Redukcia
F	Function call	bla

4 Sémantický analyzátor

5 Interpret

6 Vestavěné funkce

Vestavěné funkce jsou pro přehlednost rozděleny na dvě části, neboť podle zadání musí být funkce `find` a `sort` uloženy právě v souboru `ial.c`. Ostatní funkce jsou k nalezení v souboru `build_in.c`.

6.1 vestavěné funkce v IAL.c

–nevím jestli rozdělovat na podsekce ještě měnší nebo ne–

6.1.1 `int find(String s, String search)`

Funkce `int find(String s, String search)` hledá první výskyt zadaného podřetězce (v parametru `search`) a vrátí jeho pozici (počítanou od nuly). V naší variantě a/3/I jsme měli za úkol implementovat tuto funkci pomocí `Knuth-Morris-Prattova algorytmu`. Tato implementace spočívala v ... doplním.

6.1.2 `String sort(String s)`

Funkce `String sort(String s)` řadí parametrem daný řetězec `s` podle ordinální hodnoty obsažených znaků, který pak odevzdává návratovou hodnotou. Daný výpočet je dle zadání a/3/I zpracován pomocí řadícího algoritmu `Shell sort`, nebo též řazení se snižujícím se přírůstkem s asymptotickou složitostí $O(n^2)$. Styl implementace vychází ze vzorového příkladu přednášek předmětu IAL, kdy při prvním průchodu je brán krok jakožto polovina počtu prvků z celkové délky řetězce.

6.2 vestavěné funkce v souboru build_in.c

Pro usnadnění práce při interpretaci jsou zde u funkcí návratové typy zjednodušeny na datový typ `Value` a parametry funkcí jsou předávány pomocí datového typu `Value_list`.

- `int readInt ()` - funkce zpracovává čtený řetězec ze standardního vstupu po znacích a je zde tak zřejmá podobnost s konečnými automaty. Při nepovoleném znaku pro datový typ `int` volá funkce ke konci chybové hlášení, jinak je připuštěno k převodu řetězce na celé číslo.
- `double readDouble ()` - funkce je charakterem obdobná s funkci `readInt` a výsledná kontrola na desetinné číslo je ošetřena pomocí céčkové funkce `strtod()`.
- `String readString ()` - funkce na podobném přístupu jako `readInt` s rozšířenou množinou povolených znaků.
- `void print ( term_nebo_konkatenace )` - funkce tisknoucí svůj vstup na standartní výstup. Zde je ošetřena možnost proměnného počtu parametrů pomocí datového typu `Value_list`.

- `int length(String s)` - funkce pro zjištění délky daného řetězce, která sama vychází z céčkové funkce `strlen()`.
- `String substr(String s, int i, int n)` - funkce tiskne podřetězec daného řetězce `s` s ošetřením na přesah paměti pomocí céčkové funkce `strncat()`.
- `int compare(String s1, String s2)` - funkce porovnává dva dané řetězce na základě céčkové funkce `strcmp()`.

7 Testy

Testovali jsme buď součásti - *unit testy*, kde jsme zkoušeli, zda daná funkce správně reaguje na vstupy. Unit testy si dělal každý sám a podle potřeby. Bylo zvykem v Makefile pro unit test udělat zvláštní target, kde se kromě samotné kompilace ještě prováděl *valgrind* test pro ověření možných *memory leaků*.

Dále jsme dělali ještě systémové testy. To byly vlastně testy samotného interpretu a porovnávání jeho výstupu s výstupem Javy SE 8 s přidanou kompatibilitou s jazykem IFJ16. Test byl vytvořen jako samostatný bash script, který se volal z Makefile. Ve složce test/input/ byly různé programy v jazyce IFJ16 ve formátu *návratovýKód_názevProgramu.ifj16* s možností přidání ještě souboru se stejným formátem, ale koncovkou .in, kde byla možnost dát vstup na *stdin*. Daný skript pak prošel složku, zjistil, zda jsou v ní obsaženy i soubory typu .in pro právě interpretovaný kód. Pokud byla předpokládaná návratová hodnota 0 (chyby IFJ16 interpretu nemělo smysl překládat v Javě a porovnávat), došlo k interpretaci kódu v Javě i ifj16 interpretu s následným porovnání návratových kódů a výstupů. Vše se zapisovalo do logu, který se nacházel ve stejné složce *input* jako interpretovaný kód. Obrázek níže zobrazuje výsledky testování v průběhu raných fází interpretu.

```
ciUser@travis: ./test
Currently working on 0_ahojsvete.ifj16:
[ OK ] ... IFJ16 return code is 0, and it was expected.
[ OK ] ... IFJ16 output of 0_ahojsvete.ifj16 is correctly intepreted.

Currently working on 0_arithmetic_test_9_UNARY.ifj16:
[FAIL] ... IFJ16 return code is 2, but 0 was expected, see logs/0_fooTest.log.
[FAIL] ... IFJ16 output of 0_fooTest.ifj16 is different, see logs/0_fooTest.log.
```