VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

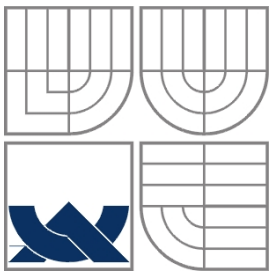
APLIKACE PRO ODHALOVÁNÍ PLAGIÁTŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

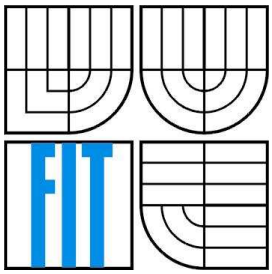
AUTOR PRÁCE
AUTHOR

PAVEL ŠALPLACHTA

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

APLIKACE PRO ODHALOVÁNÍ PLAGIÁTŮ

APPLICATION FOR DETECTING PLAGIARISM

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

PAVEL ŠALPLACHTA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ROMAN LUKÁŠ, Ph.D.

BRNO 2007

Zadání bakalářské práce

1. Prostudujte pečlivě veškeré konstrukce programovacího jazyka C.
2. Zamyslete se nad způsoby detekce dvou velmi podobných programů napsaných v jazyce C.
3. Navrhněte strukturu, která rozpozná dva potencionálně stejné programy napsané v programovacím jazyce C.
4. Danou aplikaci implementujte.
5. Funkčnost aplikace otestujte na vzorcích projektů odevzdaných studenty z minulých let.
6. Zhodnoťte dosažené výsledky, porovnejte vaši aplikaci s již existujícími aplikacemi a navrhněte další možné rozšíření do budoucna.

Licenční smlouva

Licenční smlouva je uložena v archívu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Bakalářská práce se zabývá programovacím jazykem C a způsobem odhalováním plagiátů v programech psaných v tomto jazyce. Výsledkem práce je aplikace vytvořená také v jazyce C, která podle podobnosti zdrojových kódů rozhodne, zda se jedná o plagiáty. Aplikace je především určena pro kontrolu plagiátorství domácích úloh v předmětu Algoritmy na Fakultě informačních technologií na Vysokém učení technickém v Brně.

Klíčová slova

jazyk C, plagiátorství, odhalování plagiátorství, Algoritmy, hashovací tabulka

Abstract

Bachelor’s thesis is focused on programming in language C and a method of detecting plagiarism in programs written in this language. The result of thesis is application in language C too, which according similarity source codes decides if it is a plagiarism or not. Application is first of all indicated for control plagiarism in homework in subject Algorithms at Fakulty of information technogy at Brno university of technology.

Keywords

C language, plagiarism, detection of plagiarism, Algorithms, hash table

Citace

Pavel Šalplachta: Aplikace pro odhalování plagiátů, bakalářská práce, Brno, FIT VUT v Brně, 2007

Aplikace pro odhalování plagiátů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Romana Lukáše, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Pavel Šalplachta
15.5.2007

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Romanu Lukášovi, Ph.D. za odborné vedení a podnětné rady při zpracování mé práce.

Obsah

Úvod

Na mnoha školách se můžeme setkat s tím, že studenti mezi sebou opisují domácí úlohy. Tento problém se objevuje i v domácích úkolech, kde má student vytvořit nějaký program, například v jazyce C. Jeden student program vytvoří a další si jej zkopírují a odevzdají samy za sebe, popřípadě jen něco málo v tomto programu poupraví. Toto nazýváme plagiátorstvím. Aby se dalo plagiátorství zabránit, je nejprve nutné jej odhalit a poté udělit příslušné sankce. Nejlepším způsobem odhalení plagiátorství je ruční kontrola domácích úloh všech studentů vzájemně mezi sebou. Avšak při velkém počtu studentů by je tato kontrola zabrala vyučujícím velké množství času. Nezbývá tedy nic jiného, než tuto práci automatizovat pomocí programu. A z tohoto důvodu vznikla tato bakalářská práce, která pomůže všechny plagiáty během pár vteřin odhalit.

Bakalářská práce je rozdělena do čtyř kapitol. V první kapitole se budeme zabývat programovacím jazykem C. Probereme si všechny jeho důležité konstrukce a všechny možné zápisy těchto konstrukcí. Ve druhé kapitole provedeme analýzu a navrhneme, jak potenciálně odhalit plagiáty v jazyce C. Ve třetí kapitole se zaměříme na implementační detaily aplikace pro odhalování plagiátů. Ve čtvrté kapitole budou uvedeny výsledky testování této aplikace na domácích úlohách z předmětu Algoritmy v posledních dvou letech, zhodnocení aplikace a rozšíření do budoucna.

1 Jazyk C

1.1 Úvod do jazyka C

Programovací jazyk C vyvinuli Ken Thompson a Denis M. Ritchie pro potřeby operačního systému Unix. V současné době je to jeden z nejpopulárnějších jazyků, zřejmě nejčastější pro psaní systémového softwaru, ale velmi rozšíření i pro aplikace [3].

1.1.1 Historie

„Vývoj jazyka C začal v Bellových laboratořích AT&T mezi léty 1969 a 1973. Ritchie tvrdí, že nejpřínosnější období bylo v roce 1972. Pojmenování „C“ zvolili, protože mnoho vlastností přebírali ze staršího jazyka zvaného „B“, jehož název byl zase odvozen od jazyka BCPL (ale to není jisté, neboť Thompson také vytvořil jazyk Bon na poctu své ženy Bonnie).“ [3]

1.1.2 Normy jazyka C

- **K&R** – první standard jazyka C, popsán v knize The C programming Language (autoři Brian W. Kernighan a Denis Ritchie), která vyšla v roce 1978 a kromě jiného se stala základní učebnicí jazyka C, norma určena pro systémové programátory [1]
- **ANSI C** – standart z roku 1988, vychází z K&R, avšak definuje jazyk C jako moderní vyšší programovací jazyk všeobecného použití [1]
- **ISO/IEC 9899:1999** – rozšiřující norma, která byla přijata v březnu 2000 a platí v současné době. Přináší mnoho nových vlastností např. inline funkce, pole s nekonstantní velikostí, proměnné mohou být deklarovány kdekoliv, přidány nové datové typy – long long int, bool [2]

1.1.3 Vlastnosti

„Jazyk C je nízkoúrovňový, kompilovaný, relativně minimalistický programovací jazyk. Je dostatečně mocný na většinu systémového programování (ovladače a jádro OS) a zbytek lze dořešit tzv. inline assemblerem, tedy metodou zápisu assembleru přímo do kódu. Zdrojový kód C je přitom mnohem čitelnější než assembler, je jednodušší ho zapsat a navíc je snáze přenositelný na jiné architektury. Proto jsou často operační systémy, překladače, knihovny a interprety vysokoúrovňových jazyků implementovány právě v C.

Ukládání dat je v C řešeno třemi základními způsoby: statickou alokací paměti (při překladu), automatickou alokací paměti na zásobníku a dynamickou na haldě (heap) pomocí knihovních funkcí. Jazyk nedisponuje žádnou abstrakcí nad alokací: s pamětí se pracuje přes datový typ zvaný ukazatel, který drží odkaz na paměť, ale je na něm možné provádět aritmetické operace. Ukazatelé tedy existují nezávisle na proměnných, na které odkazují, a je na odpovědnosti programátora, aby neukazovali na paměť nealokovanou.

Jazyky Java a C#, oba odvozené od C, používají méně univerzální způsob odkazování alokovaných proměnných, který snižuje pravděpodobnost chyby v programu. Jazyk C++, původně rozšíření jazyka C, si ovšem ukazatele zachoval. Mnoho dalších moderních programovacích jazyků

přebírá způsob zápisu (neboli syntaxi) z jazyka C. Patří mezi ně například zmíněná Java či Perl a PHP.“ [3]

1.2 Konstrukce jazyka C

Než začneme se tvořit aplikaci, je vhodné nastudovat všechny konstrukce jazyka C. Ty si popíšeme v následujících kapitolách.

1.2.1 Identifikátory, klíčová slova

Identifikátory jsou jména, která dáváme například **proměnným**, **funkcím** a **datovým typům**. Identifikátor se musí lišit od kteréhokoliv klíčového slova. Identifikátor je tvořen posloupností alfanumerických znaků a podtržítka, přičemž musí být splněna podmínka, že první symbol nesmí být číslice. Jazyk C je case sensitive, tzn. rozlišuje malá a velká písmena. V praxi to znamená, že: prom Prom PROM jsou tři různé identifikátory [1].

Klíčová slova mají speciální význam pro překladač C. Podle normy ANSI jsou jimi: **auto**, **break**, **case**, **char**, **const**, **continue**, **default**, **do**, **double**, **else**, **enum**, **extern**, **float**, **for**, **goto**, **if**, **inline**, **int**, **long**, **register**, **return**, **short**, **signed**, **sizeof**, **static**, **struct**, **switch**, **typedef**, **union**, **unsigned**, **void**, **volatile**, **while**. Žádný identifikátor nemůže mít ve fázi překladu stejné znění jako klíčové slovo [4].

1.2.2 Bílé znaky

Jsou jimi:

- **mezera** – značíme „ “
- **tabelátor** – značíme „\t“
- **nový řádek** – značíme „\n“
- **návrat vozíku** – značíme „\r“
- **nová stránka** – značíme „\f“

Bílé znaky spolu s operátory stojí mezi identifikátory, klíčovými slovy, řetězci a konstantami použitými ve zdrojovém textu [4].

1.2.3 Komentáře

Komentář je text, který si můžeme napsat kamkoliv do zdrojového kódu a nemá žádný vliv na překlad. Nejčastěji se používá kvůli vysvětlení určité části kódu. Vhodné používání komentářů je silně doporučeno a patří k dobré programátorské kultuře. Komentář se může vyskytnout všude tam, kde je povolen bílý znak [7].

V jazyce C jsou dva druhy komentářů:

- **Řádkové** – začínají posloupností znaků // a končí koncem řádku
- **Blokové** – jsou mezi dvojicí párových symbolů /* a */

1.2.4 Základní typy dat

Základní typy dat dělíme na **aritmetické datové typy** (celočíselné a racionální), **znaky** a na **ukazatele**.

Celočíselné:

- **int**
- **short int** (též short)
- **long int** (též long)
- **long long int** (též long long)

Racionální:

- **float**
- **double**
- **long double**

Znaky:

- **char**

Typy **char**, **short int**, **long int**, **long long int** mohou být buď typu **signed** (znaménkový) nebo **unsigned** (neznaménkový). Typ **unsigned int** se často zkracuje jen na **unsigned.** Pro celočíselné typy je implicitní typ **signed**, pro znaky záleží na implementaci [1].

Ukazatel představuje adresu paměťového místa. Jeho hodnota říká, kde je uložen nějaký objekt. Součástí deklarace ukazatele je i informace o typu dat, které jsou na získané adrese očekávány [4].

Příklad:
int *ukaz;

Hodnota *ukaz* představuje adresu paměti a **ukaz* je celočíselná hodnota uložená na této adrese.

1.2.5 Základní typy operátorů

- **Aritmetické** - sčítání (+), odečítání (-), násobení (*), dělení (/) , zbytek po dělení (%)
- **Logické** - and (&&), or (||), not (!)
- **Relační** - menší (<), větší (>), menší nebo rovno (<=), větší nebo rovno (>=), rovno (==), nerovno (!=)
- **Bitové** - posun vlevo (<<), posun vpravo (>>), bitové and (&), bitové or (|), bitové not (~), bitové xor (^)
- **Operátor přiřazení** (=)
- **Operátory inkrementace a dekrementace** – inkrementace (++) zvýší hodnotu operandu o jedničku, dekrementace (--) sníží hodnotu operandu o jedničku
- **Operátory pro práci s ukazateli** – referenční (&) a dereferenční (*)

1. Aritmetické operátory představují základní matematické operace. Výsledkem je pak podle druhů operandů buď celočíselná nebo racionální hodnota.
2. Logické operátory představují dvě hodnoty, pravda a nepravda. ANSI norma C říká, že hodnota nepravda je představována 0, zatímco pravda 1. Ve druhém případě se ovšem jedná o doporučení, neboť lze za pravdu považovat jakoukoli nenulovou hodnotu.
3. Relační operátory jsou definovány pro operandy všech základních datových typů. Jejich výsledkem jsou logické hodnoty pravda či nepravda tak, jak jsou popsány výše.
4. Bitové operátory umožňují provádět operace nad jednotlivými bity.
5. Operátor přiřazení přiřazuje proměnné hodnotu. Tento operátor lze kombinovat s aritmetickými a bitovými operátory. Příklad: x+=5; (je ekvivalentní s příkazem x=x+5;)
6. Operátory inkrementace a dekrementace se dají použít jak před svým operandem, tak za ním. Význam je však pokaždé jiný. Pokud operátor uvedeme před operandem, jedná se o tzv. *preinkrementaci*. To znamená, že hodnota operandu se nejprve zvýší a tato zvýšená hodnota je vrácena jako výsledek operace. Naopak při *postinkrementaci* je jako hodnota celého výrazu vrácena původní hodnota operandu a pak je teprve operand samotný zvýšen o 1.
7. Referenční operátor slouží k získání adresy v paměti, kde je uložená proměnná, dereferenční k získání objektu, na který ukazatel odkazuje

1.2.6 Konstanty

Konstanty jsou symboly, reprezentující neměnnou číselnou nebo jinou hodnotu. Překladač jazyka jim přiřadí typ, který této hodnotě odpovídá. Z konstant odpovídajících si typů můžeme vytvářet konstantní výrazy. Tyto výrazy musí být regulární. Nesmí obsahovat přiřazení, inkrementace a dekrementace, funkční volání, operátor čárka. Konstantám s vhodně zvolenými identifikátory dáváme přednost například před přímým uvedením konstantní hodnoty jako meze cyklu nebo dimenze pole. Modifikace programu pak probíhá velmi snadno změnou hodnoty konstanty. Odpadá obtížné uvažování, zdali ta či jiná hodnota má být modifikována či nikoliv. Konstanty mají rovněž určen typ. Tím je umožněna typová kontrola. Konstanty definujeme po klíčovém slově **const** a následovaném typem konstanty, jejím identifikátorem a po rovnítku její hodnotu ukončenou středníkem [4].

Celočíselné konstanty

V jazyce C máme tři druhy celočíselných konstant:

- **desítkové** – posloupnost číslic, z nichž první nesmí být 0 (nula)
- **osmičkové** (oktalové) – číslice 0 (nula) následovaná posloupností osmičkových číslic (0-7)
- **šestnáctkové** (hexadecimální) – číslice 0 (nula) následovaná znakem x (nebo X) a posloupností hexadecimálních číslic (0-9, a-f, A-F)

Příklady:

1, desítkové – 0, 1, 15, 90

2, osmičkové – 0, 01, 015, 065

3, šestnáctkové – 0x0, 0x1, 0xF, 0xcd, 0Xcd

Záporné konstanty označujeme znakem „-“ (mínus), například: -10. Pro konstanty typu long použijeme příponu „L“ (nebo „l“), například 123L. Další příponou je „U“ (nebo „u“), která značí unsigned, například 50u, 123LU [1].

Reálné konstanty

Tvoří se podle běžných zvyklostí, mohou začínat a končit desetinou tečkou a jsou implicitně typu double, např. 15., 5.6, .84, 5e6, 7E23.
Reálná konstanta typu float se definuje pomocí přípony F (nebo f), např. 3.14f, a typu long double pomocí L (nebo l), např. 12e3L [1].

Znakové konstanty

Hodnota znakových konstant (ordinální číslo) je odvozena z odpovídající tabulky – nejčastěji ASCII. Potřebujeme-li znakovou konstantu z neviditelného znaku, pak použijeme zápis ve tvaru ‘\ddd‘, kde ddd je kód znaku složený ze tří osmičkových číslic, např. ‘\012‘, ‘\007‘. Počáteční nuly jsou nutné. Druhou možností, jak zapsat tyto konstanty, je použití zápisu ‘\xHH‘ (nebo ‘\xHH‘), např. ‘x0A‘, ‘XD‘, ‘X1f‘.

Znak „\“ (zpětné lomítko – backsplash) se často nazývá escape charakter, protože je používán pro změnu běžného významu. Např. ‚012‘ je nedovolená konstrukce (tříznaková konstanta), protože znaková konstanta může mít jen jeden znak [1].

1.2.7 Proměnné

Proměnné jsou paměťová místa přístupná prostřednictvím identifikátoru. Hodnotu proměnných můžeme během výpočtu měnit. Tím se proměnné zásadně odlišují od konstant, které mají po celou dobu chodu programu hodnotu neměnnou – konstantní. Proměnné deklarujeme uvedením datového typu, jež je následován identifikátorem, nebo seznamem identifikátorů, navzájem oddělených čárkami. Deklarace končí středníkem. Současně s deklarací proměnné můžeme, definovat i její počáteční hodnotu [4].

Příklady deklarací proměnných:
int a, b, c , počet = 0;
float x, průměr = 0.0, odchylka;
double y;

1.2.8 Bloky

Všude v C, kde se může vyskytovat příkaz, se může vyskytovat i složený příkaz. Složený příkaz je posloupnost příkazů. Konstrukce, která složený příkaz vymezuje, začíná levou a končí pravou složenou závorkou { }, a nazývá se blok. Vnořený blok nemusí být ukončen středníkem. Představuje složený příkaz a jeho konec je jasně určen. Tělo každé funkce je také blokem [4].

1.2.9 Příkaz if - else

Příkaz *if – else* se používá, chceme-li vyjádřit rozhodování. Jeho formální syntaxe je:

if (<*vyraz*>)
  <*prikaz1*>;
else
  <prikaz2>;

Část *else* je nepovinná. Nejprve je vyhodnocen výraz „*vyraz*“. Pokud je pravdivý (má nenulovou hodnotu), je vykonán příkaz „*prikaz1*“. Je-li nepravdivý (nulový) a je-li uvedena část *else*, je vykonán příkaz „*prikaz2*“ [6].

Z předchozí kapitoly víme, že místo příkazu může být umístěn blok. V takovém případě je příkaz jasně vymezen a středník za blokem před else nepíšeme.

1.2.10 Cykly

V jazyce C můžeme použít tři příkazy pro iteraci – **while**, **for** a **do-while.** Provádění cyklů ovlivňují příkazy **break** a **continue**.

- **break** – ukončuje nejvnitřnější neuzavřenou smyčku – opouští okamžitě cyklus
- **continue** – skáče na konec nejvnitřnější neuzavřené smyčky a tím vynutí další iteraci smyčky – cyklus neopouští

Cyklus while

Tento cyklus bývá také nazýván cyklus s podmínkou na začátku. Používá se v případech, kdy dopředu neznáme počet iterací a kdy se také může stát, že cyklus nebude muset proběhnout ani jednou.

Zápis:
while (<*vyraz*>) <*prikaz*>

Výraz (podmínka) „*vyraz*“, podle kterého se rozhoduje o provádění (resp. neprovádění) iterace cyklu, se testuje ještě před prvním průchodem. Iterace se nezačne provádět, pokud je tento výraz nepravdivý. Výraz „*vyraz*“ se nazývá záhlaví cyklu, příkaz „*prikaz*“ jeho tělem. Výraz musí být uveden v závorkách [7].

Cyklus for

Tento cyklus bývá také nazýván cyklem se známým počtem průchodů. Používáme jej v případě, že už před jeho prvním průchodem víme, kolikrát se bude muset provést [7].

Zápis:
for ([<*vyraz1*>] ; [<*vyraz2*>]; [<*vyraz3*>]) <*prikaz*>

Příkaz **for** provádí příkaz „*prikaz*“ jednou, vícekrát, nebo vůbec, dokud je hodnota nepovinného testovacího výrazu „*vyraz2*“ nenulová. Výraz „*vyraz1*“ je také nepovinný. Provádí se před prvním vyhodnocením testu. Typicky se používá pro inicializaci proměnných před cyklem. Po každém provedení těla cyklu se provede opět nepovinný výraz „*vyraz3*“. Typicky se jedná o přípravu

další iterace cyklu. Pokud neuvedeme testovací výraz “ *vyraz2*“, použije překladač hodnotu 1, a bude tedy provádět nekonečný cyklus. Avšak pro jeho ukončení můžeme použít příkaz **break** [4].

Cyklus do-while

Je to jediný cyklus, který zajišťuje alespoň jedno provedení těla cyklu.

Zápis:
do <*prikaz*> while (<*vyraz*>);

Testovací výraz „*vyraz*“ je testován až po průchodu tělem cyklu. Pokud je test splněn, provádí se znovu tělo cyklu.

1.2.11 Příkaz switch (přepínač)

Přepínač slouží k rozdělení posloupností příkazů na části, následné vybrání a provedení některé, či některých z nich.

Zápis:
switch (<*vyraz*>) {
  case <*konstantniVyraz1*> : <*prikaz1*> break;
  case <*konstantniVyraz2*> : <*prikaz2*> break;
  default : <*prikaz3*> break;
}

Po klíčovém slově **switch** následuje celočíselný výraz „*vyraz*“ uzavřený v kulatých závorkách. Za ním zpravidla následuje blok, který představuje posloupnost příkazů, v níž mohou být umístěna návěští. Řídící příkaz tvoří celočíselný konstantní výraz uvozeným klíčovým slovem **case** a ukončený dvojtečkou. Jedno z návěští může být klíčovým slovem **default**, přirozeně ukončeným dvojtečkou. Nastane-li shoda hodnoty **case** s hodnotou switch výrazu „*vyraz*“, je přeneseno řízení programu na toto návěští. Pokud nenastane shoda s žádným výrazem case, je řízení přeneseno na návěští **default** [4].

1.2.12 Funkce

Jsou základními stavebními kameny jazyka C. Je ekvivalentní podprogramu. Představuje výhodný způsob zapouzdření nějakého výpočtu, aniž bychom se museli později starat o jeho implementaci. V jazyce C je používání funkcí jednoduché, šikovné a efektivní. Často narazíme na krátkou funkci, která je pouze jednou definovaná a pouze jednou zavolaná, jenom proto, že to zjednodušuje chápání části programu [6].

Deklarace funkce:
navratovy_typ jmeno_funkce ([seznam_argumentu]);

Definice funkce:
```
navratovy_typ jmeno_funkce ([seznam_argumentu])
{
  // telo funkce
}
```

- *navratovy_typ* – nejsou na něj kladena žádná omezení. Pokud funkce nevrací žádnou hodnotu, použije se klíčové slovo **void**. Návratová hodnota se vrátí pomocí argumentu v příkazu **return**, který se dává do těla funkce.
- *seznam_argumentu* - je nepovinný. Funkce nemusí mít žádné argumenty, může mít jeden nebo více argumentů, nebo také můžeme určit, že funkce má proměnný počet argumentů [2]. Deklarace všech formálních argumentů funkcí musí obsahovat datový typ. Není možné uvést společný typ pro více následujících identifikátorů.

Funkci voláme tak, že napíšeme název (identifikátor) funkce a do kulatých závorek dáme seznam předávaných argumentů.

Příklady volání funkce:
```
printf (“ahoj“);
prumer=vypocti_prumer(5.5);
```

1.2.13 Pole

Pole je kolekce proměnných stejného typu, které mohou být označovány společným jménem. K prvkům pole přistupujeme prostřednictvím identifikátoru pole a indexu. Pole v C obsazuje spojitou oblast operační paměti tak, že první prvek je umístěn na nejnižší přidělené adrese a poslední na nejvyšší přidělené adrese [4].

Definice:
```
typ jmeno[rozsah];
```

Typ určuje typ prvků pole, *jmeno* představuje identifikátor pole a *rozsah* je kladný počet prvků pole (celočíselný konstantní výraz, který musí být vyčíslitelný za překladu).
V jazyce C je možné deklarovat pouze jednorozměrné pole. Jeho prvky ovšem mohou být libovolného typu. Mohou tedy být opět jednorozměrnými poli. To však již dostáváme vektor vektorů, tedy matici.

Definice dvourozměrného pole:
```
typ jmeno [rozsah1][rozsah2];
```

1.2.14 Struktura, union

Zatímco pole je homogenní datový typ – všechny jeho prvky jsou stejného typu – struktura je datový typ heterogenní (datový typ je složen z datových prvků různých typů, což je ale jeho vnitřní záležitostí, protože navenek vystupuje jako jednolitý objekt).

Definice:
```
struct {
   int vyska;
   float vaha;
} pavel, honza, karel;
```

Další možnosti definice struktury jsou:
```
stuct miry {
  int vyska;
  float vaha;
};
struct miry pavel;
struct miry honza, karel;
```

```
typedef struct {
  int vyska;
  float vaha;
} MYRI;
MYRI pavel, honza, karel;
```

Ještě další možností je ještě za **typedef stuct** přidat název struktury. Tato možnost se používá, když chceme, aby struktura mohla odkazovat sama na sebe. To se využívá například u spojovaných seznamů.

Přístup k prvkům struktury je pomocí tečkové notace:
```
pavel.vyska=175;
```

Pokud použijeme ukazatel na strukturu (např. MYRI *pavel;), můžeme k prvkům přistupovat dvěma způsoby:
```
1. (*pavel).vyska=175;
2. pavel->vyska=175;
```

Podobnou syntaxi jako má struktura, má i **union**, který je obdobou pro **variantní záznam** v programovacím jazyce Pascal. Datový typ union znamená, že se vyhradí paměť pro největší položku ze všech položek v unionu definovaných. Všechny položky unionu se překrývají (ve struktuře by ležely v paměti za sebou), což znamená, že v něm může být v jednom okamžiku pouze jedna položka. V praxi se uniony používají málokdy [1].

1.2.15 Výčtový typ (enum)

Výčtový typ má podobnou konstrukci jako struktura či union, ale má zcela jinou filosofii práce. Pomocí něho lze snadno definovat seznam symbolických konstant, které mohou být vzájemně závislé. Podobně jako u struktury a unionu má i výčtový typ několik různých způsobů zápisu.

Nejčastějším zápisem je:

```
typedef enum {
 MODRA, CERVENA, ZELENA, ZLUTA
} BARVY;

Barvy c,d;
C = MODRA;
D = CERVENA;
```

Pokud explicitně nepřiřadíme číselné hodnoty jednotlivým prvkům vyjmenovaného typu, pak mají tyto prvky implicitní hodnoty 0, 1, 2. Chceme-li jiné hodnoty, použijeme tento zápis:

```
typedef enum {
 MODRA=0, CERVENA= 4,  ZELENA=2 , ZLUTA
} BARVY;
```

Barva ZLUTA není inicializována, má tedy hodnotu 3 [1].

1.2.16 Příkaz skoku - goto

Zápis:
navesti: <*prikaz*>;
goto <*navesti*>;

Návěští „*navesti*“ neprovádí žádnou akci a nemá jiný vliv na řízení běhu. Jeho identifikátor jen označuje nějaké místo ve zdrojovém textu. Příkaz goto přenáší řízení na příkaz, označený návěštím. V rámci jedné funkce nesmí být dvě návěští stejná.

Příkaz skoku se využívá málokdy, protože ve strukturovaných programech se bez něj lze vždycky obejít. Udává se, že nenapomáhá dobré orientaci v programu a jeho použití většinou nevrhá zrovna dobré světlo na programátora. Avšak existují situace, ve kterých je jeho použití vhodné, např. vyskočení z hlubokého zanoření v cyklech [7].

1.2.17 Podmíněný operátor

Vedle podmíněného příkazu if-else existuje v jazyce C ještě další způsob, jak podmínky zapsat. A to pomocí podmíněného operátoru, který má tři operandy. Nazývá se proto ternární operátor.

Zápis:
vyraz1 ? *vyraz2* : *vyraz3*

Nejdříve je vyhodnocen *vyraz1*, který by měl být číselného typu. Je-li jeho hodnota nenulová, je následně vyhodnocen *vyraz2* a jeho hodnota je zároveň výslednou hodnotou celého výrazu. *Vyraz3* se vůbec nevyhodnotí a tudíž se neprovedou ani případné postranní efekty. V případě, že je *vyraz1* vyhodnocen jako nula, je naopak vyhodnocen *vyraz3* a jeho hodnota je také výsledkem celé operace [5].

1.2.18 Operátor postupného vyhodnocování

Tento operátor se také nazývá operátor **čárka**. Je jeden z mála operátorů, který zajišťuje pořadí vyhodnocení svých operandů.

Zápis:
vyraz1, vyraz2

Nejdříve je vyhodnocen levý operand (*vyraz1*), jehož hodnota je ovšem zapomenuta, a který slouží především k vykonání postranních efektů. Jako druhý je pak vyhodnocen *vyraz2*, jehož hodnota je pak i výslednou hodnotou celého výrazu. Operátor čárky se často využívá například v řídících částech příkazu for, kde jeho použití může vést ke zjednodušení zápisu, nebo kde slouží jako prostředek k vykonání dalších postranních efektů [5].

Příklad:
for (i=0, j=4, i<5, i++, j--)

1.2.19 Preprocesor jazyka C

Preprocesor ještě před samotným překladem zdrojového souboru tento soubor upravuje. Činnost preprocesoru řídíme pomocí tzv. direktiv preprocesoru. Každá direktiva je uvozena znakem „#“, který musí být uveden hned jako první znak na řádku. Tak určíme, že zbytek řádku je určen preprocesoru a zápisy v něm se tedy řídí jeho syntaktickými pravidly, která nejsou totožná s těmi céčkovskými. Příkazy preprocesoru lze je rozdělit na více řádků pomocí znaku ‚\‘.

Direktiva #define

Tato direktiva se používá pro vytváření tzv. maker, které slouží k zpřehlednění zdrojového kódu programu. Použitím této direktivy říkáme preprocesoru, aby každý následující výskyt identifikátoru makra ve zdrojovém souboru nahradil textem makra. Výjimkou jsou ale ty výskyty identifikátoru makra, které se nacházejí v řetězcích a poznámkách [5].

Zápis:
#define makro_id(*argumenty*) [*sekvencePrikazu*]

makro_id – identifikátor (jméno) makra
argumenty – nepovinný seznam argumentů makra navzájem oddělených čárkou
sekvencePrikazu – nepovinný souvislý řetězec, kterým se nahradí výskyt řetězce *macro_id* ve zdrojovém kódu

Příklady maker:
#define POCET_PRVKU 128
#define SQR(x) (x*x)
#define max(a,b) ((a>b)?a:b)

Direktiva #include

Je naprosto nepostradatelnou direktivou. Používá se pro včlenění zdrojového textu jiného souboru. Tento soubor může být určen dvěma způsoby:

- #include <*jmeno_hlavicky*> – soubor *jmeno_hlavicky* je hledán ve standardním adresáři pro include. Takto se zpravida začleňují standardní hlavičkové soubory.
- #include "*jmeno_hlavicky*" - soubor *jmeno_hlavicky* je hledán v pracovním adresáři. Není-li tam nalezen, postupuje se podle předchozí možnosti. Takto se zpravidla začleňují uživatelské hlavičkové soubory.

Příklady:
#include <stdio.h>
#include "htable.h"

Direktivy pro podmíněný překlad:

Preprocesor může během své činnosti vyhodnocovat, je-li nějaké makro definováno či nikoliv. Při použití klíčového slova preprocesoru **defined** pak můžeme spojovat taková vyhodnocení do rozsáhlejších logických výrazů. Argument defined nemusí být uzavřen do závorek. Může se však vyskytnout jen za direktivami #**if** nebo #**elif**. V závislosti na splnění či nesplnění podmínky můžeme určit, bude-li ohraničený úsek programu zpracován, nebo bude-li odfiltrován a tak nebude tedy přeložen. Aby bylo jasno, kde podmíněná část zdrojového textu začíná a kde končí, nesmíme zapomínat na #**endif** či #**elif**. Výraz #if defined lze zkrátit na #**ifdef**. Další direktivou je #**ifndef**, která je obdobou #ifdef s tím rozdílem, že následující kód se provede, pokud makro následující za #ifndef není definováno. Poslední direktivou pro podmíněný překlad je #**else**. Kód za direktivou se provede v tom případě, že nebyla splněna žádná předchozí podmínka [4].

Příklad:
```
#define LADENI
#ifdef LADENI
  printf ("Provadime ladeni programu");
#endif;
```

Ostatní direktivy

- #**error** – je direktivou, kterou můžeme zajistit výstup námi zadaného chybového hlášení
- #**line** – umožňuje nastavit hodnotu standardního makra __LINE__ a případě i __FILE__. Používá se zejména u strojově generovaných zdrojových textů
- #**pragma** – je speciální direktivou, která má uvozovat všechny implementačně závislé direktivy. Pokud jiný překladač speciální direktivu nezná, prostě ji bez chybového stavu ignoruje
- #**undef** – oddefinování symbolické konstanty

1.2.20 Další klíčová slova

auto

Dává se před definici datového typu. Definuje lokální proměnnou.

extern

Dává se před definici datového typu. Definuje globální proměnnou (automaticky inicializovaná na 0, pokud to programátor neudělá jinak). Samotná direktiva se používá k referenci na globální proměnnou uvnitř funkce (zvláště hrozí-li záměna).

inline

Toto klíčové slovo se dává před deklaraci funkce. Používá se jen pro velmi krátké funkce, které jsou relativně často volané. Dává se tím překladači najevo, že se kód té funkce má zkopírovat pokaždé na místo, kde je volán. Dosáhne se tím určitého zrychlení programu za cenu zvětšení jeho kódu.

register

Dává se před definici datového typu. Umístí proměnnou do registru procesoru, je-li to možné, jinak ignoruje. Je možné použít jen pro lokální proměnné.

sizeof (*datový typ*)

Vrací velikost daného datového typu. Místo slova označující daný datový typ se dá použít proměnná tohoto datového typu (se stejným výsledkem).

static

Dává se před definici datového typu. Definuje se jako lokální proměnná, chová se jako globální proměnná.

volatile

Klíčové slovo *volatile* má stejnou syntaxi a podobný význam jako *const* s tím rozdílem, že *volatile* může být změněno pomocí některých systémových přerušení.

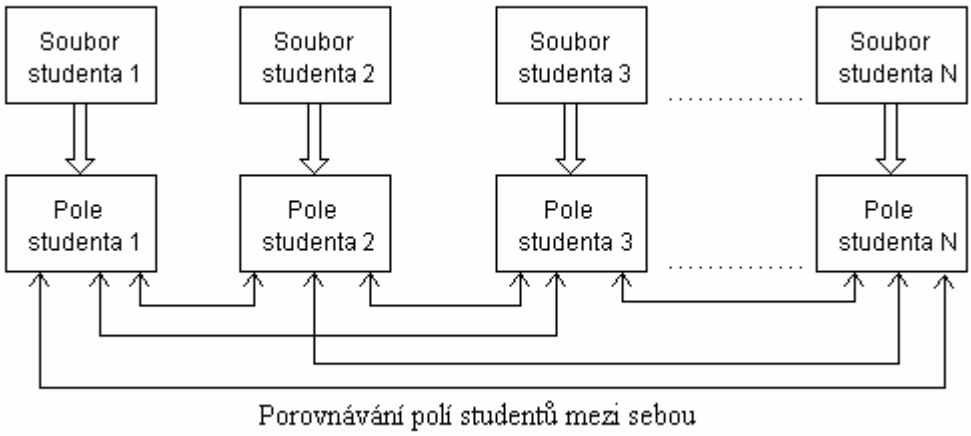
2 Analýza daného problému

2.1 Způsob detekce

V předchozí kapitole jsme se dozvěděli o možných konstrukcích jazyka C. Nyní můžeme navrhnout, jak dva potenciálně stejné programy můžeme odhalit. Je nutné zvolit co nejrychlejší možnost, protože pokud se bude například kontrolovat 500 studentů, musí se provést:

$$\binom{n}{k} = \binom{500}{2} = \frac{500 * 499}{2} = 124750$$

Musí se tedy provést 124750 porovnání. Proto rozdělíme kontrolu na dvě části. V první části vytvoříme pole čísel. Všem prvkům pole nastavíme hodnotu 0. Poté zpracujeme jednotlivé zdrojové kódy všech studentů, zjistíme počet konstrukcí a nakonec pole čísel každého studenta naplníme počtem jednotlivých konstrukcí jazyka C, které má ve svém zdrojovém kódu. V druhé části budeme mezi sebou porovnávat pouze pole čísel. Pro každé dva studenty zjistíme celkový počet rozdílů mezi jednotlivými konstrukcemi a pokud bude menší, než stanovená dolní hranice, budou označeny jako plagiáty. Tímto dosáhneme velké rychlosti kontroly, neboť se v každém cyklu se porovná pouze pole celých čísel.



Porovnávání polí studentů mezi sebou

2.2 Předzpracování jednotlivých zdrojových kódů

Než začneme rozpoznávat jednotlivé konstrukce jazyka C, je vhodné načíst zdrojový kód do textového řetězce, se kterým se lépe pracuje, než se souborem. Dále je potřeba odstranit všechny nepotřebné elementy. Jsou jimi řádkové komentáře, blokové komentáře a nadbytečné bílé znaky. To je nutné z tohoto důvodu, že nijak neovlivňují běh programu. A také odstranění těchto nadbytečností ulehčí rozpoznávání konstrukcí jazyka C. Odstranění provedeme tak, že postupně budeme načítat znaky z jednoho řetězce a do druhého je uložíme bez komentářů, nadbytečných bílých znaků a dalších elementů.

Řádkový komentář rozpoznáme tak, že začíná posloupností znaků „//“. Jakmile narazíme na tyto znaky, přestaneme do výstupního řetězce znaky ukládat, dokud nenarazíme na konec řádku („\n“). Pro **blokové komentáře** to uděláme podobně s tím, že začínají posloupností znaků „/*“. Následující znaky se ignorují, dokud se nenarazí na posloupnost znaků „*/“. Při procházení komentáři (jak řádkovými, tak i blokovými) si zapamatujeme počet jejich znaků. To bude sloužit k tomu, abychom mohli poté určit, zda student komentoval zdrojový kód, neboť v předmětu Algoritmy [8] je povinné domácí úlohy komentovat. Aby však nedocházelo ke zkreslení, nebudeme započítávat všechny bílé znaky v komentářích.

Z **bílých znaků** se ve zdrojovém kódu mohou nacházet: mezera, nový řádek, návrat vozíku, tabelátor a nová stránka. Mezi příkazy lze bíle znaky bez problému vynechat. Uvnitř příkazu lze vynechat bílé znaky mezi operátory, závorkami atd. Je mnoho případů, kdy lze bílé znaky vynechat, zaměříme se tedy na případy, kdy je nelze vynechat. To je mezi dvěmi identifikátory, klíčovými slovy či dvěma čísly. A tak když narazíme na nějaký bílý znak, zapamatujeme si poslední znak vložený do výstupního řetězce, poté procházíme řetězcem tak dlouho, dokud nenarazíme na jiný znak, než bílý. A teď pokud je tento znak znakem identifikátoru (číslice, malé či velké písmeno nebo podtržítko) a zároveň zapamatovaný znak (před bílými znaky) je též znakem identifikátoru, vloží se mezi ně mezera. Pokud v jazyce C použijeme znak dolar („$“), chová se tento znak stejně jako podtržítko, tudíž v názvu identifikátoru se může nacházet i tento znak, přestože se o něm v učebnicích jazyka C nepíše. A tak identifikátor budeme definovat tak, že začíná znakem anglické abecedy, podtržítkem či dolarem a za ním mohou následovat stejné znaky a číslice.

Vhodné je i odstranění textu v **uvozovkách**, neboť se tím usnadní další zpracování, navíc text v uvozovkách se většinou používá u funkce „prinft“ či podobných funkcí pro výpis nějakého textu a ty budeme ignorovat (viz níže). Odstranění textu je jednoduché, pokud narazíme na znak uvozovek, ignorují se další znaky, dokud se nenarazí na další znak uvozovek. Avšak je nutné dát pozor na to, zda se před tímto znakem nenachází zpětné lomítko („\“) , které ruší platnost následujícího znaku. V tomto případě načítáme a ignorujeme znaky dále, dokud nenarazíme na uvozovky, před kterými není zpětné lomítko.

Toto je vhodné provést i mezi **apostrofy** (‘‘). Mezi apostrofy se většinou nachází pouze jeden znak, který lze nahradit i číselnou konstantou. Ve zvláštních případech (viz kapitola 1.2.13) se mezi apostrofy nachází i více znaků (většinou speciální znaky), které lze rovněž nahradit číselnou konstantou. A tak znaky mezi apostrofy odstraníme a při rozpoznávání konstrukcí pokud se narazí na apostrofy, nahradí se znakem pro číslo (viz následující kapitola). Znaky odstraníme stejným způsobem jako v případě textu mezi uvozovkami.

Celý zdrojový kód je vhodné před rozpoznáváním konstrukcí rozdělit na menší celky. Nejlepší možnost je rozdělit jej po jednotlivých příkazech, tj. po středníkách. Avšak nesmíme zapomenout na příkazy preprocesoru, které nekončí středníkem (například **makra).** Ty končí s koncem řádku. Musíme brát i v úvahu makra na více řádků a tedy pokud narazíme na zpětné lomítko, načítáme makro dál. Zpětné lomítko neukládáme do výstupního řetězce. Nyní už nic nebrání rozpoznávání konstrukcí jazyka C.

2.3 Rozpoznávání konstrukcí jazyka C

Nyní je nutno rozpoznat identifikátory, klíčová slova, operátory a další konstrukce ze vstupního řetězce. Do výstupního řetězce uložíme znaky odpovídající jednotlivým konstrukcím.

2.3.1 Identifikátory, klíčová slova

Jak již je zřejmé z kapitoly 1.2.1, identifikátory se poznají tak, že začínají malým nebo velkým písmenem anglické abecedy, podtržítkem nebo dolarem (dolar - viz předchozí kapitola 2.2) a následující znaky navíc mohou obsahovat i číslice. Identifikátor uložíme do pomocného řetězce, který postupně porovnáme s jednotlivými klíčovými slovy jazyka C a hlavními vestavěnými funkcemi. Pokud se bude s některým klíčovým slovem či vestavěnou funkcí shodovat, do výstupního řetězce vložíme symbol, který označuje tuto konstrukci a budeme dál načítat znaky ze vstupního řetězce. V opačném případě se bude jednat o název funkce, datový typ nebo proměnnou. Ještě, než se zaměříme na tyto tři možnosti, probereme si ještě některé vestavěné symbolické konstanty, funkce, klíčová slova a direktivy preprocesoru.

Symbolické konstanty „**EXIT_SUCCESS“**, „**EXIT_FAILURE“**, „**true“**, „**false“**, „**TRUE“**, „**FALSE“** mají hodnotu nula nebo jedna. Tudíž je můžeme nahradit číslem.

Vestavěné funkce „**printf**“, „**fprinft**“ můžeme ignorovat i s jejich obsahem, neboť z našeho pohledu nezáleží na tom, co se vypisuje. Funkcím pro alokaci paměti (**malloc**, **realloc**) a uvolnění paměti (**free**) přiřadíme vlastní čísla konstrukce, neboť jsou v domácích úlohách v předmětu Algoritmy [8] často používané. Naopak klíčová slova „**auto**“, „**extern**“, „**register**“, „**static**“ a „**volatile**“ se prakticky nevyskytují, tudíž je vůbec nemusíme brát v úvahu.

Direktivy preprocesoru

Direktivy preprocesoru (začínají znakem #) můžeme až na makra (#define) ignorovat. Pro makro uložíme do výstupního řetězce příslušný znak. Při ignoraci klíčového slova direktivy ignorujeme i všechny další znaky za tímto slovem. Zamezíme tím například toho, že nedojde ke změně počtu konstrukcí, když se přidají do zdrojového souboru nadbytečné direktivy #include <*nazev.h*>. Dále se tím může zamezit změně počtu konstrukcí, když se například přidá podmíněný překlad.

Příklad:
```
#ifdef DEBUG
printf (“Obsah proměnné i je %d.“, i);
#endif
```

Protože budeme direktivy preprocesoru ignorovat a při podmíněném překladu se většinou vypisuje obsah nějaké proměnné a funkci „printf“ také ignorujeme, neprojeví se podmíněný překlad v počtu konstrukcí.

Nyní se budeme zabývat rozpoznáváním názvů **funkcí**, **datových typů** a **proměnných**. **Ukazatel** (referenční i dereferenční) budeme pro zjednodušení uvažovat jako normální proměnnou.

Název funkce

To, že identifikátor značí název funkce poznáme tak, že za ním následuje otevřená kulatá závorka. Obecně v jazyce C může na názvem funkce následovat mezera a pak až kulatá otevřená závorka, ale jak již bylo zmíněno v předchozí kapitole 2.2, všechny nadbytečné bílé znaky byly před rozpoznáváním konstrukcí odstraněny. Po rozpoznání funkce ještě zjistíme, jestli se jedná o volání funkce nebo deklaraci funkce. To se zjistí podle toho, zda je před názvem funkce datový typ. Takže se podíváme do výstupního řetězce na poslední uložený znak a pokud je jím znak, odpovídající datovému typu, jedná se o deklaraci funkce. Jinak je to volání funkce. Kromě znaku odpovídající

deklaraci či volání funkce vložíme ještě do výstupního řetězce znak otevřené kulaté závorky, která však bude mít pouze pomocný charakter při dalším rozpoznávání konstrukcí.

Datový typ

Datový typ lze rozpoznat podle toho, že za ním následuje mezera (v obecném případě i tabelátor, nový řádek, více mezer či kombinace těchto možností, avšak všechny tyto možné případy byly nahrazeny jednou mezerou – viz kapitola 2.2), hvězdička nebo ampersand a před ním se nachází: složená závorka (otevřená či uzavřená), otevřená kulatá závorka nebo je to první identifikátor v řetězci. Tento způsob zjištění datového typu je výhodný tím, že se rozpoznají i nově definované datové typy. Je důležité ještě myslet na datové typy **long long int**, **long int** a **long double**, které jsou složeny z více identifikátorů. Toto lze vyřešit tím, že pokud se narazí na long, zapamatujeme si to a poté když následuje identifikátor double, long či int, ignorujeme ho, neboť to, že jde o datový typ se již zaznamenalo, když se narazilo na první long. Dále existuje datový typ **short int**, který lze zapsat zkráceně jako **short** (viz kapitola 1.2.4). Vyřešíme to stejně jako s datovými typy long long int, long int, long double. Dále lze ještě zkracovat typ **unsigned int** na **unsigned**, překladač též vezme pouze datový typ **signed** (bez toho, aniž by za ním byl například datový typ int, long apod.) Proto pokud narazíme na klíčové slovo **signed** či **unsigned**, zaznamenáme si to (stejně jako u typů short int, long int, atd.) a pokud se poté narazí na klíčové slovo **int**, **long** nebo **short**, ignorujeme je. Nyní se budou deklarace např. **unsigned short int x;** a **short x;** brát jako totožné. Avšak musíme se zaměřit ještě na jeden problém. A to, pokud za znakem „*“ nebo „&“ následuje znak „=“. V tom případě se nejedná o datový typ, za kterým následuje ukazatel, nýbrž o násobní (pro hvězdičku) nebo bitový and (pro ampersand) proměnné s výrazem za rovnítkem. Proto do výstupního řetězce uložíme přiřazení proměnné, použití proměnné a aritmetický (v případě hvězdičky) nebo bitový (v případě ampersandu) operátor. To z tohoto důvodu, abychom nahradily např. konstrukci x*=10 konstrukcí x=x*10.

Proměnná

Pokud se nebude jednat o název funkce ani datový typ, jedná se o proměnnou. **Symbolickou konstantu** budeme také považovat za proměnnou. Rozlišíme, zda se jedná o deklaraci proměnné či její použití. Deklarace se pozná podle toho, že poslední uložený znak ve výstupním řetězci je datový typ. Sice ještě předním by měla být mezera, hvězdička či ampersand, to vyřešíme tak, že po datovým typu tuto mezeru, hvězdičku či ampersand ignorujeme, do výstupního řetězce nedáváme. Jedná li se tedy o deklaraci proměnné, znak značící datový typ přepíšeme znakem značící deklaraci proměnné. O deklaraci proměnné se také jedná, pokud se proměnná nachází za deklarací struktury. Takže pokud byl jako poslední vložený znak do výstupního řetězce zavřená složená závorka (uzavírací závorka deklarace struktury) a aktuální vstupní znak je čárka či koncová nula, vložíme do výstupního řetězce znak, který odpovídá deklaraci proměnné.

Dále nesmíme zapomenout na operátor **čárka**. Pokud se před proměnnou nachází čárka, zjistíme, zda se ještě před čárkou nachází deklarace proměnné. V kladném případě se jedná též o deklaraci proměnné. V opačném případě se může jednat o čárku mezi parametry ve funkci, ale též může jít o to, že předchozí proměnné je přiřazena též nějaká hodnota. Proto musíme projít celý vstupní řetězec a zjistit, zda se v něm nenachází deklarace proměnné. Před procházením výstupního řetězce musíme si uložit index, kde se aktuálně nacházíme a pak od něj zase pokračovat. Pokud během prohledávání narazíme na nějakou složenou závorku, můžeme přestat v prohledávání, neboť se již nacházíme mimo příkaz (to se děje v případě, že součástí příkazu je i například hlavička funkce

či cyklus => nenachází se za nimi středník, proto se zpracovávají i s prvním příkazem). Najdeme-li znak znamenající deklaraci proměnné, vložíme do výstupního řetězce opět znak deklarace proměnné. Pokud se nebude jednat o deklaraci proměnné, vložíme do výstupního řetězce znak použití proměnné.

Struktura

Ještě je vhodné zaměřit se na datový typ **struktura** (resp. i union či výčtový typ, které však mají podobnou syntaxi, tudíž klíčové slovo struct, union a enum můžeme označit stejným symbolem). Z kapitoly 1.2.14 víme, že strukturu lze definovat několika syntakticky různými způsoby. Proto označíme při deklaraci proměnné pomocí struktury klíčové slovo struct (resp. union či enum) a název struktury jako datový typ. To tak, že pokud se ve výstupním řetězci jako poslední znak mezera, před ním je znak struktury a aktuální vstupní znak je také mezera, přepíšeme znak struktury znakem datového typu. A dále při definici struktury nebudeme ukládat její název, tzn. pokud je aktuální znak vstupního řetězce otevřená složená závorka a jako poslední znak výstupního řetězce je mezera a přední znak struktury, budeme identifikátor ignorovat. Tím dosáhneme toho, že následující 3 příklady budou brány stejně:
1. struct {*polozky*} prom1, prom2;
2. struct *nazev* {*polozky}* prom1, prom2;
3. struct *nazev* {*polozky}*; struct *nazev* prom1; struct *nazev* prom2;

Tímto jsou vyřešené všechny identifikátory. Nyní se zaměříme na ostatní konstrukce v jazyce C.

2.3.2 Čísla

Posloupnost číslic je též vhodné shrnout do jednoho symbolu. Pokud tedy narazíme na číslici (0-9) nebo tečku, za kterou následuje číslice (0.5 <==> .5, viz kapitola 1.2.6) načítáme další, dokud nenarazíme na něco jiného, než číslici, tečku nebo malé či velké písmeno „e“. V případě písmena *e* se jedná o exponenciální číslo. Za znakem e se může nacházet znak „+“ nebo „-“. V tom případě jej načteme a pak opět načítáme číslice od nuly do devíti. Pokud se za číslem nějaký ze znaků (popřípadě i dva) „u“, „U“, „l“ , „L“, „f“, „F“ (viz kapitola 1.2.6) načteme je také. Nakonec do výstupního řetězce uložíme znak značící číslo. Narazíme-li na dva apostrofy za sebou (znaky mezi apostrofy byly zrušeny v předzpracování), vložíme také znak značící číslo.

Musíme však ještě uvažovat šestnáctková čísla. Ty začínají nulou a za ní následuje malé x nebo velké X a v číslu se mohou nacházet znaky a-f či A-F. Do výstupního řetězce uložíme znak značící číslo, neboť je zbytečné od sebe odlišovat desítková a šestnáctková čísla.

Osmičková čísla nemusíme uvažovat, neboť ty začínají nulou a vyskytují se v nich číslice 0-7, což se zpracovává již u desítkových čísel.

2.3.3 Operátory

Relační operátory (>, <, <=, >=, ==, !=) označíme jedním znakem, protože tím odhalíme obrácení podmínky. Abychom zamezily také negaci podmínky, **logické operátory** (||, &&) označíme stejným znakem a znak **!** ignorujeme.

Nyní se např. odhalí případy:
if (a>0) prikaz; <==> (a>=1) prikaz;

if (a!=0) prikaz1; else prikaz2; <==> if (a==0) prikaz2; else prikaz1;

Aritmetické operátory (+, -, *, /, %) označíme jedním symbolem, stejně tak **bitové** (>>, <<, ~ , ^, |, &). Pokud za aritmetickým nebo bitovým operátorem následuje znak „=“, vložíme ještě do výstupního řetězce znak „použití proměnné“. To z toho důvodu, abychom například nahradily příkaz x+=1 příkazem x=x+1.Tuto změnu nemusíme provádět pro násobení a bitový and, neboť jsme to zpracovali již při rozpoznávání datového typu.

Operátory * a & se musíme důkladně analyzovat a určit, zda se jedná o násobení (resp. bitový and) a nebo o dereferenční (resp. referenční) ukazatel. Ukazatele uvažujeme jako proměnné, * a & ve formě ukazatele můžeme ignorovat, v opačném případě je označíme jako aritmetický (resp. bitový) operátor. O ukazatel se jedná, pokud je hvězdička (resp. ampersand) jako první symbol nebo se nachází za :

- znakem „=“
- složenou závorkou
- klíčovým slovem else či return
- otevřenou kulatou závorkou
- operátorem čárky
- logickým nebo relačním operátorem

Poslední možností je, že se nachází za uzavřenou kulatou závorkou, avšak pouze v případě, že před závorkou, kterou uzavírá se nachází příkaz if, while, for. Avšak narazíme na další problém v případě příkazu for. Rozpoznávání se provádí pouze po příkazech rozdělenými středníky (u preprocesoru koncem řádku). V cyklu for se v hlavičce argumenty rozdělují středníkem. Ale to, že se jedná o cyklus for můžeme zjistit tak, že nenarazíme ve vstupním řetězci na otevírací kulatou závorku. Pro hvězdičku je ještě jedna možnost, že za ní následuje další hvězdička (resp. hvězdičky) => násobná dereference. Pokud tato možnost nastane, ignorujeme všechny hvězdičky.

Operátory ++ (**inkrementace**) a -- (**dekrementace**) se nahradí znaky přiřazení proměnné, aritmetický operátor a číslo. To z toho důvodu, že nahradíme například x++ za x=x+1. Znak použití proměnné se již v řetězci nachází.

2.3.4 Pole

Pokud narazíme na otevřenou hranatou závorku, bude se jednat o **pole**. Zjistíme, co se nachází před hranatou závorkou. Jedná-li se o použití proměnné, nahradí se znakem označující použití pole, deklarace proměnné se nahradí deklarací pole, přístup k prvku struktury se nahradí přístupem k prvku pole struktury.

2.3.5 Operátor přiřazení („=“)

Pokud narazíme na znak „=“, zjistíme, zda je jako poslední uložený znak deklarace proměnné, v tom případě vložíme do výstupního řetězce znak odpovídající přiřazení proměnné k hodnotě, pokud je znak použití proměnné, tak tento znak ve výstupním řetězci přepíšeme znakem přiřazení proměnné. Tím se nám podaří odhalit často prováděnou úpravu zdrojového kódu při plagování: „int x=5;“ a „int x; x=5;“. Dále pokud se před „=“ nachází znak přístup k prvku struktury, nahradíme jej znakem přiřazení prvku struktury. V případě, že se před „=“ nachází uzavřená hranatá závorka „]“, projdeme výstupním řetězcem, dokud nenarazíme na odpovídající otevřenou hranatou závorku „[“. Nyní

zjistíme, jaká konstrukce se nachází před touto závorkou. Je-li jí použití pole, nahradíme jí konstrukcí přiřazení hodnoty prvku k pole, použití pole ve struktuře nahradíme přiřazení hodnoty pole ve struktuře, přístup k prvku struktury nahradíme přiřazením prvku ve struktuře.

2.3.6 Další symboly

Narazí-li se na posloupnost znaků „-“ a „>“, nahradí se jedním znakem značící -> (přístup k prvku struktury). Dále můžeme ignorovat kulaté závorky, protože pokud byly potřeba, byly zpracována již dříve, v tomto případě jsou nadbytečné. Ignorovat také můžeme zpětné lomítko (znak „\“), který by se však již neměl v příkazu nacházet. Objevit se zde může případě, že se při předzpracování v makru nacházel středník. Nakonec pokud se narazí na nějaký jiný znak, uloží se do výstupního řetězce. Tím znakem může být například # (zpracovává se až s identifikátorem za tímto znakem), mezera či čárka.

2.4 Výpočet počtu konstrukcí

Nyní již máme textový řetězec, ve kterém se nacházejí ascii znaky, které odpovídají jednotlivým konstrukcím jazyka C. Dále máme pole čísel, které má výchozí hodnoty samé nuly. Nyní toto pole naplníme počtem konstrukcí daného zdrojového kódu. To tím způsobem, že každého ascii znaku z textového řetězce zjistíme jeho ordinální hodnotu, a v poli čísel inkrementujeme číslo, které se nachází na indexu této ordinální hodnoty. Nesmíme však zapomenout nezapočítávat znaky „{“, „}“, „(“, „)“, „[“, „]“, „=“, „!“ a mezeru, neboť ty jsou již zpracované a ve výstupním řetězci měli pouze pomocnou úlohu při rozpoznávání konstrukcí.

2.5 Porovnávání polí čísel

Teď už máme pole čísel počtu konstrukcí. Nyní postupně porovnáme tyto pole studentů mezi sebou. Jak již bylo spočítáno v kapitole 3.1, pro 500 studentů se musí těchto porovnání provést 124750. Porovnávat budeme tak, že nejprve vezmeme pole prvního studenta a to porovnáme s polem druhého studenta. Pokud zjistíme, že se jedná o plagiát, vypíšeme login prvního studenta a login druhého studenta. Login druhého studenta si zaznamenáme. To z tohoto důvodu, když například bude plagovat skupina několika lidí, aby se členové skupiny vypsali jen jednou (pokud se bude jednat o plagiát, zjistí se, zda je login studenta již vypsán a v tomto případě se už nevypisuje). Dále se bude postupně porovnávat pole prvního studenta se všemi ostatními. Potom se vezme pole druhého studenta, které se bude postupně porovnávat s polem třetího až posledního studenta. Tak to bude pokračovat, až se nakonec pole předposlední studenta zkontroluje s polem posledního studenta.

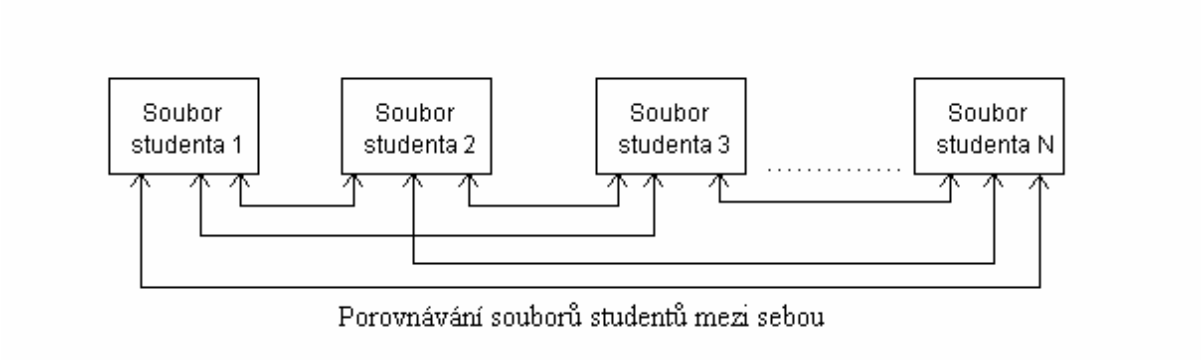
Nyní již víme, jak budeme pole jednotlivých studentů kontrolovat mezi sebou. Zbývá nám ještě určit, jak zjistíme pomocí dvou polí čísel, že studenti, kterým tyto pole přísluší, mezi sebou plagovali. To provedeme tak, že postupně budeme porovnávat jednotlivá čísla pole. Pokud budou rozdílná, zjistíme jejich absolutní rozdíl a připočteme jej k celkovému počtu rozdílů. Poté pokud bude celkový počet rozdílů menší, než bude stanovená dolní hranice, bude se jednat o plagiáty. Dolní hranici buď stanový uživatel programu, nebo se může určit tak, že se zjistí nejmenší počet rozdílů od referenčního řešení.

2.6 Další přísnosti aplikace

Nyní již máme aplikaci navrhnutou. Avšak pokud studenti budou mít za úkol doplnit například jen několik řádků kódu, kde není moc možností, vypíše se i na celkový počet rozdílů 0 velké množství plagiátů, které však nelze považovat za plagiáty. Proto je vhodné přidat další testování s menší přísností. Přísnosti začínaly od nuly až po N, kde číslo přísnosti značí celkový počet rozdílů mezi dvěma zdrojovými kódy. Pro snadné odlišení budou nově přidané přísnosti značeny zápornými čísly a to od -4 po -1.

2.6.1 Přísnost -4 (absolutní shoda)

V tomto testu pouze načteme zdrojové kódy jednotlivých studentů do řetězce a pak je bez jakýkoliv úprav porovnáme mezi sebou. Porovnávání mezi studenty proběhne stejně, avšak místo polí čísel se mezi sebou budou porovnávat řetězce. Na tento test se moc plagiátů neodhalí, avšak ty, co se odhalí, můžeme 100% považovat za plagiáty. Porovnání řetězců bude trvat delší dobu, avšak protože neprobíhá žádné zpracování kódu, proběhne i tento test plagiátorství velmi rychle.



Porovnávání souborů studentů mezi sebou

2.6.2 Přísnost -3 (velmi malá přísnost)

Postup stejný jako v předchozím testu, akorát po načtení zdrojových kódů do řetězce ještě provede jejich předzpracování, tzn. při porovnávání se nebudou brát v úvahu komentáře, nadbytečné bílé znaky atd.(viz kapitola 2.2).

2.6.3 Přísnost -2 (malá přísnost)

Oproti předchozí přísnosti se provede z části rozpoznávání konstrukcí, avšak zpracují se pouze identifikátory, klíčová slova a čísla. Takže se odhalí vzájemně přejmenované proměnné, funkce a různé datové typy.

2.6.4 Přísnost -1 (střední přísnost)

Proběhne stejně jako u testu malá přísnost, avšak místo porovnávání řetězců se řetězec každého studenta převede na pole počtu výskytů jednotlivých znaků a budou se porovnávat tato pole. Tím se navíc odhalí přeházení příkazů, podmínek atd. U této přísnosti nebudeme oproti přísnostem 0 a vyšší ignorovat znaky „=“ a „!“, neboť nejsou zpracovány.

3 Implementační detaily

3.1 Funkce „zpracovani“

Tato funkce zpracovává text zdrojového kódu. Výstupem je pak buď pole počtu výskytů jednotlivých konstrukcí jazyka C a nebo zpracovaný řetězec. Funkce dále počítá počet znaků v komentářích.

Parametry funkce jsou:

- **vstupní řetězec** – zdrojový kód studenta
- **pole čísel konstrukcí** – pole počtu jednotlivých konstrukcí jazyka C, které se ve zdrojovém kódu nachází, používá se jen pro střední a velkou přísnost
- **výstupní řetězec –** zpracovaný řetězec, používá se jen pro velmi malou a malou přísnost
- **ukazatel na počet znaků komentářů** – pomocí tohoto parametru předáváme počet znaků v komentářích
- **přísnost** – pomocí tohoto parametru zjišťujeme, co se má všechno má funkce provádět
- **pole hashovacích tabulek** - předává se funkci „rozpoznani“

Podle přísnosti se určí, zda výstupem funkce bude zpracovaný řetězec nebo pole počtu výskytů konstrukcí jazyka C. Funkce vrací 0, pokud proběhne vše bez problémů, v případě chyby (například neuzavřený blokový komentář, neuzavřené uvozovky, nelze nealokovat paměť) vrací příslušné číslo chyby. Funkce ke své činnosti využívá funkcí „**rozpoznani**“, která přejmenovává konstrukce na ascii znaky, a „**vlozeni_do_pole**“, která převádí řetězec ascii znaků na pole počtu výskytů jednotlivých ascii znaků. Avšak tyto funkci nejsou využity vždy, záleží na hodnotě přísnosti.

Na začátku se deklarují a inicializují důležité proměnné. Jsou jimi index vstupního textu (**index=0**), ukazatel na pomocný řetězec (**pom_retezec=NULL**), počáteční velikost pomocného řetězce (**velikost=0**), index pomocného řetězce (**index_pom=0**), pomocná booleovská proměnná pro uložení stavu, že se jedná o příkazy preprocesoru (**preproc=false**), počet znaků komentářů (**komentar=0**). Po inicializaci se prochází v cyklu vstupním řetězcem, dokud se nenarazí na ukončující znak „\0“. Cyklu se nachází několik podmínek, podle kterých se určí, co se má s daným znakem provést (viz kapitola 2.2).

Narazí-li se na posloupnost znaků „//“, v cyklu, který končí, pokud se narazí na znak „\n“ nebo na znak „\0“ (komentář na posledním řádku zdrojového kódu), se pro každý procházení znak (kromě bílého znaku) inkrementuje proměnná „*komentar*“.

Pro posloupnost znaků „/*“ se v cyklu, který končí, pokud se narazí na posloupnost znaků „*/“, se opět pro každý procházení znak (kromě bílého) inkrementuje proměnná „*komentar*“. Avšak pokud se během procházení narazí na znak „\0“, uvolní se paměť a funkce vrací číslo, značící chybu „neuzavřený blokový komentář“.

Pokud se narazí na jiný znak, v cyklu se postupně načítají tyto znaky a ukládají do pomocného řetězce „**pom_retezec**“, který se alokuje dynamicky. Jestliže proměnná, určující aktuální pozici v tomto řetězci, je větší než je aktuální alokovaná velikost řetězce, zvětší se velikost o 128 a zároveň se alokuje paměť pro dalších 128 znaků pomocného řetězce. Narazí-li se na znak „#“, poznamenáme si tuto informaci do proměnné preprocesoru (*preproc=true*). Načítání do této proměnné trvá tak dlouho, dokud se nenarazí na nějaký bílý znak, středník, apostrof, uvozovky, lomítko, či koncovou nulu. Nyní se podle tohoto znaku vyhodnocuje dál.

Narazí-li se na lomítko („/“), znamená to, že se jedná buď o dělení, nebo řádkový či blokový komentář. O komentář se bude jednat, pokud je následující znak ve vstupním řetězci lomítko („/“) nebo hvězdička („*“). V tom případě se vracíme do hlavního cyklu této funkce. V opačném případě se jedná o dělení a tak tento znak dělení vložíme do pomocného řetězce. Až poté se vracíme do hlavního cyklu funkce.

Jedná-li se o *uvozovky*, vloží se pomocného řetězce a pak následující znaky se ignorují, dokud se nenarazí opět na další uvozovky, před kterými však není znak „\“. Uvozovky se vloží do pomocného řetězce a vracíme se do hlavního cyklu. Pokud však místo uzavíracích uvozovek narazíme na koncovou nulu (text v uvozovkách není ukončen dalšími uvozovkami), uvolní se paměť a funkce vrátí chybovou hlášku značící chybu „Neuzavřené uvozovky“. V případě *apostrofu* platí stejný postup, jako v případě uvozovek.

Další možností je, že můžeme narazit na nějaký bílý znak uvnitř příkazu. V tomto případě si do pomocné proměnné „predchozi“ uložíme poslední znak vložený do pomocného řetězce a ze vstupního řetězce načítáme v cyklu další znaky, dokud nenarazíme na jiný, než bílý znak. Všechny tyto bílé znaky, kromě znaku nový řádek („\n“), nebereme v úvahu. V případě, že narazíme na konec řádku a proměnná „preproc“ je rovna hodnotě true (viz předchozí odstavec) a poslední znak před bílými znaky není „\“, jedná se o direktivu preprocesoru. Proměnné „preproc“ přiřadíme hodnotu false a pomocné proměnné „predchozi“ přiřadíme středník, aby byla na konci hlavního cyklu tato direktiva zpracována (viz níže). Pokud ale je poslední znak před bílými znaky roven znaku „\“, vymažeme tento znak z pomocného řetězce (index pomocného řetězce snížíme o jedničku) a pokračujeme v cyklu, který probíhá, dokud se nenarazí na jiný než bílý znak. Poté pokud byl poslední znak před bílými znaky (proměnná „predchozi“) a zároveň první znak za bílými znaky roven některému ze znaků patřící identifikátoru (a-z, A-Z, 0-9, _, $), vložíme do pomocného řetězce znak mezery. V opačném případě do něj nic nevkládáme.

Nyní ještě se otestuje, zda aktuální znak na vstupu je *středník* či *koncová nula* (popřípadě proměnná „predchozi“ je rovna středníku – direktiva preprocesoru). Pokud ano (v případě, že aktuální znak na vstupu je středník, zvýšíme ještě aktuální pozici indexu vstupního řetězce), vytvoří se dvě pomocné proměnné (pole znaků velikosti indexu pomocného řetězce). Do prvního se zkopíruje obsah pomocného řetězce a vloží na konec řetězce koncová nula. Potom se podle hodnoty proměnné „**prisnost**“ rozhodne, co se provede (buď se pouze pomocný řetězec připojí k výslednému řetězci, nebo se zavolá funkce „**rozpoznani**“, které předáme pomocný řetězec, druhý pomocný řetězec (prázdný) a jako třetí parametr hodnotu true (funkce zpracuje pouze identifikátory, klíčová slova a čísla). Poté se druhý pomocný řetězec připojí k výslednému. Další možnost je, že po vrácení druhého pomocného řetězce zavoláme funkce „**vlozeni_do_pole“**, které předáme tento řetězec a výsledné pole čísel. Poslední možností je, že se zavolá funkce rozpoznání s parametrem false (rozpoznávají se všechny konstrukce) a pak se zavolá funkce „**vlozeni_do_pole**“.

Nyní se vynuluje proměnná, která určuje aktuální pozici v pomocného řetězce a vrátíme se do hlavního cyklu. Po zpracování celého vstupního řetězce se uvolní paměť pomocného řetězce a funkce vrátí 0 (proběhla v pořádku).

3.2 Funkce „rozpoznani“

Tato funkce je největší a nejsložitější na celém programu. Převádí upravený zdrojový kód na ascii znaky, kde každý ascii znak představuje nějakou konstrukci či klíčové slovo jazyka C.

Parametry funkce jsou:

- **vstupní řetězec** – upravený zdrojový kód bez komentářů, nadbytečných bílých znaků, textu v uvozovkách atd.
- **výstupní řetězec** – do tohoto řetězce se uloží ascii znaky jednotlivých konstrukcí jazyka C
- **booleovská proměnná „cast“**, která určuje, jestli se má při rozpoznávání zpracovávat pouze identifikátory a klíčová slova(využito u malé a střední přísnosti) nebo všechny konstrukce jazyka C (využívá se pro přísnosti od 0 a vyšší)
- **pole hashovacích tabulek** – 3 tabulky, ve kterých se budou vyhledávat klíčová slova či vestavěné funkce (v první tabulce se nachází vestavěné funkce a klíčová slova, za kterými následuje otevřená kulatá závorka, v druhé tabulce ostatní klíčová slova a vestavěné symbolické konstanty a ve třetí tabulce direktivy preprocesoru)

Funkce využívá funkci **htHledani**, které předává konkrétní hashovací tabulku a klíč, který chce v tabulce vyhledat a funkce vrátí hodnotu tohoto klíče. Pokud zadaný klíč nenajde, vrací NULL. Dále se využívá **symbolických konstant**, ve kterých jsou definovány názvy jednotlivých konstrukcí jazyka C. Díky nim si nemusíme například pamatovat, že klíčové slovo while patří znaku ordinální hodnoty 2, ale stačí napsat lépe zapamatovatelnou symbolickou konstantu „KLICOVE_while“.

Funkce nejprve deklaruje a inicializuje pomocné proměnné, kterými jsou: index vstupního řetězce (**index=0**), index výstupního řetězce (**index2=0**), pomocný řetězec (**pom_retezec**) velikosti vstupního řetězce, do kterého se budou ukládávat názvy identifikátorů/klíčových slov, index pomocného řetězce (**index_pom=0**), pomocná proměnné položka hashovací tabulky (**prom**) a 3 booleovské proměnné pro určování složených datových typů (**longtyp=false**, **signed_unsigned=false**, **shortint=false**).

Nyní se prochází vstupním řetězcem, dokud se nenarazí na koncový znak „\0“. V tomto cyklu se nachází velké množství podmínek, pomocí kterých se určuje daná konstrukce jazyka C.

Jako první se zpracovávají **identifikátory** (resp. klíčová slova). Název identifikátoru se uloží do pomocného řetězce. Nyní se zjišťuje, jestli se jedná o proměnnou, klíčové slovo, název funkce nebo datový typ.

Pokud následuje za identifikátorem otevřená kulatá závorka, zavolá se funkce htHledani, které předáme jako parametr první hashovací tabulku obsahující **klíčová slova**, která mají za sebou závorku (např. if, while, for, switch) a **vestavěné funkce** (např. prinft, exit, malloc). Pokud funkce nevrátí NULL a vrácená data jsou různá od symbolické konstanty „IGNORACE“, vloží se do výstupního řetězce vrácená data (příslušný ascii znak). Pokud vrátí ignoraci, načítáme znaky ze vstupního řetězce, dokud nenarazíme na odpovídající uzavírací závorku. Všechny tyto znaky zahodíme. Název klíčového slova (resp. funkce) také nikam neukládáme. V případě, že vyhledávací funkce vrátila NULL, zjistíme, zda se nachází ve výstupním řetězci jako poslední znak, který odpovídá datovému typu. V kladném případě tento znak se přepíše jiným, který odpovídá deklaraci funkce, jinak ve do výstupního řetězce vloží znak, odpovídající volání funkce. Do výstupního řetězce vložíme také otevřenou kulatou závorku, která bude sloužit jako pomocný znak při rozpoznávání dalších konstrukcí a později bude ignorována. Závorku vložíme vždy, když se nachází za klíčovým slovem nebo se jedná o název funkce.

V případě, že se za řetězcem nenachází levá otevřená závorka, zpracovávají se **složené datové typy**. Je-li poslední zpracovaný identifikátor „long“ (zjistíme tak, že proměnná „longtyp==true“) a obsah pomocného řetězce je roven řetězci „long“, „int“ nebo „double“, inkrementuje se pouze index

vstupního řetězce, do výstupního řetězce se nic nevkládá. To stejné se provede, pokud poslední zpracovaní identifikátor byl „singed“ či „unsigned“ (signed_unsigned==true) a obsah pomocného řetězce je roven řetězci „int“, „char“, „short“ nebo „long“, popřípadě pokud poslední zpracovaný identifikátor byl „short“ (shortint==true) a obsah pomocného řetězce je roven řetězci „int“. Tím jsou zpracovány složené datové typy. Zde se nevyužívá hashovací tabulky, neboť je potřeba také vědět hodnotu pomocných proměnných a proto je tento princip výhodnější.

Nejedná-li se o složený datový typ, rozpoznávají **direktivy preprocesoru**. Ty se poznají tak, že poslední uložený znak ve výstupním řetězci byl „#“. Zavoláme funkci „htHledani“ pro třetí hashovací tabulku. Pokud vrátí NULL, znamená to, že znak # byl využit pro jinou činnost (například spojování proměnných) a tak pokračujeme další podmínkou. V opačném případě pokud jsou vrácená data různé od ignorace, vložíme je do výstupního řetězce, jinak index2 dáme rovný nule a příkazem break ukončíme hlavní cyklus. Za hlavním cyklem se totiž do výstupního řetězce vkládá na pozici index2 znak „\0“, tudíž funkce vrátí prázdný řetězec.

Jestliže se nejednalo o příkaz procesoru, rozpoznávají **ostatní klíčová slova** a vestavěné symbolické konstanty. Zavolá se tedy funkce „htHledani“ pro druhou tabulku. Pokud vrátí hodnotu různou od NULL a data jsou různá od ignorace, vloží se do výstupního řetězce data, v případě ignorace se pouze inkrementuje index vstupního řetězce.

V případě, že se nejednalo o žádnou z předchozích možností, identifikátor bude buď datový typ nebo proměnná. Rozpoznání a zpracování nesloženého **datového typu** se provede dle analýzy (viz kapitola 2.3.1).

Dále se zjistí, zda je před identifikátorem uzavírací složená závorka a za ním čárka nebo koncová nula („\0“). V tom případě vložíme do výstupního řetězce ascii znak odpovídající deklaraci proměnné (jedná se o deklaraci proměnné za strukturou). Jinak se nejedná o datový typ a tudíž vložíme do výstupního řetězce ascii znak odpovídající použití proměnné.

Při plném zpracování funkce proběhne ještě podrobnější zpracování proměnných. Jestliže je poslední znak výstupního řetězce roven ascii znaku, který odpovídá datovému typu, přepíšeme jej znakem deklarace proměnné. Je-li poslední znak výstupního řetězce **čárka** a před ním je deklarace proměnné, přepíšeme čárku znakem deklarace proměnné (čárka zůstává zachována pouze u argumentů funkce). V opačné případě prohledáme výstupní řetězec, jestli se v něm nachází znak deklarace proměnné. Pokud ano, přepíšeme čárku znakem deklarace proměnné v opačném případě vložíme do řetězce znak použití proměnné. Posledním zpracováním je zpracování struktur (resp. unionů a výčtových typů). Postup je také popsán v kapitole 2.3.1. Pokud se neprovede žádná s podmínek, vloží se do výstupního řetězce znak použití proměnné.

Zbývá ještě zpracovat složené datové typy. Ty se sice už zpracovávaly (viz výše), avšak je třeba ještě nastavit pomocné booleovské proměnné (longtyp, shortint, signed_unsigned). Tuto sekci provádíme pokaždé, když se zpracovávají identifikátory (resp. klíčová slova) bez ohledu na to, zda se už provedla některá z podmínek. Takže porovnáme obsah pomocného řetězce s řetězcem „long“, pokud jsou shodné, nastavíme proměnnou longtyp=true, v opačném případě longtyp=false. To stejné provedeme pro řetězec „short“ (nastavení proměnné shortint) a společně pro signed a unsigned (nastavení proměnné signed_unsigned).

Tím jsou všechny identifikátory zpracovány. Index pomocného řetězce nastavíme na hodnotu 0. Nyní následují další podmínky, které se zpracovávají v případě, že ve vstupním řetězci se nachází nějaký jiný znak. Pro částečné zpracování funkce (malá a střední přísnost) se zpracovávají pouze čísla (desítkové i šestnáctkové), pokud se jedná o jiný znak, vloží se do výstupního řetězce. V plném zpracování funkce navíc probíhá zpracování všech operátorů, polí, operátoru přiřazení, závorek atd.

To se provádí stejným postupem, který je uveden v analýze (viz kapitoly 2.3.3 až 2.3.6). Pokud se bude jednat o jiný symbol, vložíme jej do výstupního řetězce.

Pro proběhnutí celého cyklu (jak již bylo zmíněno výše) se do výstupního řetězce na pozici proměnné index2 vloží znak "\0".

3.3 Funkce „porovnani"

Funkce porovná jednotlivá pole čísel nebo řetězce všech studentů a podle přísnosti vypíše loginy studentů, které vyhodnotí jako plagiáty.
Parametry funkce jsou:

- **pole polí čísel jednotlivých konstrukcí** – pole, která budeme mezi sebou porovnávat – používá se pro střední a velkou přísnost
- **pole řetězců** – pole řetězců, která budeme mezi sebou porovnávat – používá se pro absolutní shodu, velmi malou a malou přísnost
- **struktura STUDENT** – pro zjištění loginu studenta
- **přísnost –** pomocí ní určujeme, zda se budou porovnávat řetězce nebo pole čísel
- počet studentů

Funkce nejprve inicializuje proměnné. Podle přísnosti zjistí, zda mezi sebou bude porovnávat pole čísel nebo řetězce. Pro pole čísel se pomocné booleovské proměnné „**cisla**" nastaví hodnota true, pro řetězce hodnota false. Poté v cyklu for (označíme jej *vnější cyklus*) probíhá porovnávání. Počet cyklů bude o jeden kratší, než je hodnota načtených loginů. Nyní se zjistí, zda je první hodnota pole čísel aktuálního studenta různá od nuly (pro porovnávání řetězců zda řetězec aktuálního studenta je roven NULL). V tom případě se cyklus pro tohoto studenta ukončí (nebude se kontrolovat – neodevzdal nebo je chyba v souboru) a pokračuje se dalším studentem. V opačném případě se začne probíhat další cyklus for (označíme jej vnitřní cyklus), který začíná od studenta o jednoho dál, než je ve vnějším cyklu a končí posledním studentem. Nyní se opět zjistí, zda se student bude kontrolovat. V kladném případě se pro pole čísel zavolá funkce „**porovnani_poli**", které se předá pole aktuálního studenta ve vnějším cyklu, pole aktuálního studenta vnitřního cyklu a přísnost. Pokud funkce vrátí -1 jedná se o plagiáty. Pro řetězce se zavolá vestavěná funkce „**strcmp"**, které se předají řetězce aktuálního studenta vnějšího a vnitřního cyklu. Pokud jsou řetězce shodné, vrací funkce hodnotu 0. Pokud se jedná o plagiáty, zjistíme, zda není některý z těchto studentů již vypsán, tzn. projdeme pomocné pole čísel „**uz_vypsani**", zda se v něm nachází čísla těchto studentů. Pokud ne, vypíšeme loginy těchto studentů (v případě, že již aktuální student z vnějšího cyklu byl již v tomto cyklu jednou vypsán, vypíše se pouze login aktuálního studenta z vnitřního cyklu) a číslo studenta vnitřního cyklu vložíme do pomocného pole čísel „**uz_vypsani**".

3.4 Funkce pro práci s hashovací tabulkou

3.4.1 Funkce „hashFunkce"

Funkci předáme klíč (řetězec) a počet řádků tabulky, na základě toho vrátí index klíče v tabulce. Výpočet indexu provede tak, že sečte ordinální hodnoty všech znaků klíče, vydělí je počtem řádku klíče a index do tabulky je zbytkem po tomto dělení.

3.4.2 Funkce „htInicializace“

Funkci předáme ukazatel na tabulku (struktura) a počet řádků, které má mít. Inicializace proběhne tak, že se do struktury tabulky uloží počet jejich řádků, alokuje se paměť (počet řádků krát velikost položky tabulky) a poté se jednotlivým položkám řádku přiřadí hodnota NULL.

3.4.3 Funkce „htHledani“

Funkci předáme hashovací tabulku (struktura) a klíč (řetězec), který chceme v tabulce vyhledat. Nejprve se zavolá „hashFunkce“, která vrátí index klíče (řádek v tabulce). Poté na tomto řádku procházíme jednotlivé položky, pokud je klíč položky roven danému klíči, vrátí se ukazatel na tuto položku. Pokud se daný klíč v tabulce nenajde, vrátí funkce NULL.

3.4.4 Funkce „htVlozeni“

Funkci předáme tabulku (struktura), klíč (řetězec) a data (znak). Pomocí funkce „htHledani“ zjistíme, zda se daný klíč už v tabulce nachází. V tom případě pouze přepíšeme data. V opačném případě alokujeme paměť (velikost položky tabulky), položce tabulky přiřadíme klíč a data, pomocí „hashFunkce“ zjistíme index klíče a na tuto pozici dáme vytvořenou položku. Za ni připojíme seznam dalších položek, které se nacházeli na stejné pozici indexu klíče.

3.4.5 Funkce „htSmazaniVsechRadku“

Je poslední funkcí pro práci s hashovací tabulkou. Zruší všechny položky na všech řádcích tabulky. To provede tak, že postupně prochází řádky tabulky, uvolní paměť jednotlivých položek a první položce na řádku přiřadí NULL.

3.5 Funkce „vytvoreni_tabulek“

Vstupem je pole hashovacích tabulek. Funkce tyto tabulky inicializuje a vloží do nich odpovídající data. Modifikací této funkce lze změnit, která klíčová slova, funkce či příkazy preprocesoru se mají ignorovat, jak se mají chápat atd. Do první tabulky (index 0) se vkládají klíčová slova, za kterými následuje závorka a funkce, které má program podporovat. Do druhé tabulky (index 1) se vkládají ostatní klíčová slova a konstanty (např. true, false, EXIT_FAILURE, EXIT_SUCCESS atd.) A nakonec do třetí (index 2) se vkládají příkazy preprocesoru. Využívá se funkcí „htInicializace“ a „htVlozeni“. Dále se využívá symbolických konstant pro vkládání dat do tabulky.

Způsob vkládání do tabulky:
htVlozeni(tabulka[čís*lo tabulky*], *klíč*, *hodnota*);

Příklady vložení:
htVlozeni (tabulka[0],“if“, KLICOVE_if);
htVlozeni (tabulka[0],“printf“, IGNORACE);
htVlozeni (tabulka[1], “EXIT_FAILURE“, CISLO);

U klíčových slov, za kterými může avšak nemusí být kulatá otevřená závorka se musí vložit do první (index 0) i do druhé tabulky (index 1). Jsou jimi například klíčová slova **return** a **case**.

Data do jednotlivých tabulek jsou vloženy na základě analýzy (viz kapitola 2.3.1). Pokud se bude v budoucnu tato funkce nějak modifikovat, je vhodné také změnit velikosti jednotlivých tabulek. Současné velikosti tabulek (18, 27, 14) byly zvoleny tak, aby byly položky vložené do tabulky umístěny co nejefektivněji a zároveň aby tabulka zabírala co nejméně paměti.

Dále je vhodné zdůraznit, že klíčová slova, funkce a příkazy preprocesoru, které se vyskytují nejčastěji je dobré vkládat jako poslední, aby se při vyhledávání v tabulce našli rychleji (to z toho důvodu, že se vkládá na začátek seznamu).

3.6 Funkce „hlavni1“

Tato funkce řídí aplikaci pro přísnost absolutní shoda, velmi malá přísnost a malá přísnost.

Parametry funkce jsou:

- **pole struktur STUDENT** (obsahuje login studenta a počet znaků komentářů v jeho zdrojovém kódu)
- **počet studentů** (celé číslo)
- **jméno testovaného souboru** (řetězec)
- **cesta k testovanému souboru** (řetězec)
- **přísnost** (celé číslo)
- **minimální počet znaků komentářů** (celé číslo)

Pokud proběhne bez problému vrací 0 (BEZ_CHYBY). V případě chyby (může nastat chyba při alokaci paměti) vrací 1 (CHYBA_ALOKACE_PAMETI).

Funkce nejprve inicializuje proměnné. Nejdůležitější z nich je pole řetězců s názvem „pole“. Pro malou přísnost se navíc vytvoří hashovací tabulky (zavolá funkci „vytvoreni_tabulek“). Poté následuje cyklus, ve kterém se postupně zpracují zdrojové kódy jednotlivých studentů. To probíhá tak, že nejprve se otevře zdrojový soubor studenta. Pokud se jej však nepodaří otevřít (student nejspíš soubor neodevzdal), vypíše se hláška, že u tohoto studenta se nepodařilo soubor otevřít a v proměnné „pole“ se na index právě zpracovávaného studenta přiřadí NULL, což znamená, že se nebude kontrolovat s ostatními a začne se zpracovávat následující student. V případě úspěšného otevření zdrojového kódu se pokračuje v jeho zpracování. Zjistí se velikost souboru, která se nám bude hodit k tomu, abychom věděli, kolik se má později alokovat paměti pro příslušný zdrojový kód. O dalším pokračování rozhoduje přísnost. U přísnosti absolutní shoda se pouze podle velikosti souboru alokuje na příslušném indexu v poli paměť a načte se do ní obsah zdrojového kódu. Pro přísnosti velmi malá a malá přísnost platí následující postup. Obsah zdrojového kódu se načte do pomocného řetězce a zavolá se funkce „zpracovani“, které se předají parametry pomocný řetězec, na místo pole polí čísel se dá NULL, dále prázdný řetězec, který funkce naplní výslednými znaky, referenci proměnné položky struktury STUDENT „delka_komentaru“, přísnost a pole hashovacích tabulek. Pro velmi malou přísnost jsou tyto tabulky prázdné, neboť nebudou použity. Pokud funkce „zpracovani“ skončí bez chyby, alokuje se na příslušném indexu pole řetězců paměť velikosti výstupního řetězce funkce „zpracovani“ a obsah výstupního řetězce se tam zkopíruje. V případě, že skončí s chybou z důvodu, že se nepodařilo alokovat paměť, ukončí se i tato funkce (vrátí 1 – chyba alokace paměti). V případě

jiné chyby (neuzavřený blokový komentář, neuzavřené uvozovky, neuzavřené apostrofy) se vypíše příslušná chyba studenta a v proměnné „pole“ se na index právě zpracovávaného studenta přiřadí NULL. Na konci každého cyklu se uzavře soubor (zdrojový kód) studenta.

Po skončení cyklu se pro malou přísnost zruší hashovací tabulky. Poté se zavolá funkce „porovnani“, která porovná pole řetězců všech studentů mezi sebou a vypíše plagiáty. Pokud skončí chybou alokace paměti, ukončí se i tato funkce. Dále se už může uvolnit paměť, kterou zabírají pole řetězců. Pro přísnost absolutní shoda funkce končí a vrací se 0 (proběhla bez chyby). Pro další dvě přísnosti se navíc ještě v případě, že minimálního počtu znaků komentářů je nezáporný, vypíše pro každého studenta počet znaků v komentářích. Pokud existuje i referenční řešení (popřípadě kostra programu), které má název „x“ (název lze měnit změnou hodnoty symbolické konstanty „REFERENCNI“) a je umístěno v souboru s loginy na prvním místě, odečte se od počtu komentářů studenta počet komentářů v referenčním řešení a pokud je menší nebo rovno minimálnímu počtu znaku komentářů, vypíše se tento rozdíl. Pokud není referenční řešení, počet komentářů referenčního řešení se neodečítá. Po vypsání počtu znaků komentářů vrátí funkce 0 (proběhla bez chyby).

3.7 Funkce „hlavni2“

Funkce je podobná funkci „hlavni1“. Rozdílem je, že nepracuje s polem řetězců, ale s polem polí celých čísel. Toto pole se inicializuje na začátku funkce a hodnoty se nastaví na nulu. Cyklus zpracování probíhá obdobě jako pro přísnosti velmi malá a malá přísnost, avšak funkci „zpracovani“ předáme místo pole řetězců NULL a na místo, kde má být parametr pole polí čísel dáme námi vytvořené na začátku funkce (ve funkci „hlavni1“ jsme předávali NULL). Další změnou je, že pokud se zdrojový kód studenta nebude kontrolovat, vloží se na první pozici pole studenta hodnota různá od nuly. Následně při volání funkce „porovnani“ se předává vytvořené pole polí čísel a na místo parametru pole řetězců se dá NULL.

3.8 Funkce „main“

Tato funkce se spouští jako první při spuštění aplikace. Má parametry argc, což je počet argumentů příkazové řádky a argv, což je pole řetězců – argumenty příkazové řádky.

Pokud je počet argumentů různý od pěti či šesti, vypíše se chybová hláška. V opačném případě se tyto argumenty příkazové řádky zpracují a uloží do proměnných. Vytvoříme strukturu „STUDENT“, která obsahuje položky „login“ (řetězec osmi znaků) a „delka_komentaru“ (celé číslo). Nyní se zavolá funkce „nacteni_loginu“, které předáme název souboru s loginy a referenci na proměnnou „počet_lidi“, pomocí které zjistíme počet načtených loginů. Funkce nám dále pro každou strukturu v poli alokuje paměť a do položky „login“ vloží login studenta. Pokud funkce vrátí NULL, ukončí se celá aplikace (vrací 1 - skončila s chybou).

Poté se na základě přísnosti zavolá buď funkce „hlavni1“ (pro přísnost -4 (absolutní shoda), -3 (velmi malá přísnost) a -2 (malá přísnost)) nebo funkce „hlavni2“ (pro ostatní přísnosti -1 (střední přísnost) a přísnosti rovny či větší, než 0 (velká přísnost)). Když vybraná funkce skončí bez chyby, uvolní se paměť vyhrazená pro pole struktury STUDENT a ukončí se aplikace (vrátí 0 – proběhla v pořádku), v případě chyby se rovnou aplikace ukončí (vrací 1 – skončila chybou).

V případě, že přísnost byla zadána v určitém intervalu, provádí se volání funkce „hlavni1“ či „hlavni2“ (dle aktuální přísnosti) v cyklu pro všechny přísnosti v tomto intervalu.

3.9 Ostatní funkce

3.9.1 Funkce „vlozeni_do_pole“

Vstupem funkce je řetězec a pole čísel. Funkce prochází řetězcem a podle ordinální hodnoty každého znaku se v poli čísel na pozici této ordinální hodnoty inkrementuje číslo. Přitom se ignorují kulaté závorky a mezery. Hranaté a složené závorky není třeba ignorovat, neboť počet prvků pole je 64 a ascii znaky se od ordinální hodnoty 64 automaticky ignorují. Avšak v budoucnu při zvětšení pole čísel (změna hodnoty symbolické konstanty „VELIKOST_POLE“) je třeba brát tyto hranaté (91, 93) a složené závorky (123, 125) v úvahu.

3.9.2 Funkce „porovnani_poli“

Vstupem této funkce jsou dvě pole čísel a přísnost. Návratová hodnota je celkový počet rozdílů mezi všemi čísli pole nebo -1. Prostupně projde všemi prvky pole a pokud se nějaké prvky liší, přičte se absolutní hodnota jejich rozdílu k celkovému počtu rozdílů. Pro přísnost větší než -1 se ignorují znaky „!“ a „=“. Pokud je celkový počet rozdílů větší než je přísnost, vrátí funkce hodnotu -1. Pokud se po průchodu polem nedosáhne celkového počtu rozdílů většího než přísnost, vrátí funkce tento celkový počet rozdílů.

3.9.3 Funkce „nacteni_loginu“

Vstupem funkce je řetězec, ve kterém je název souboru, ve kterém se nachází loginy studentů a ukazatel na číslo s počtem těchto loginů. Funkce vrací ukazatel na pole struktur STUDENT, do kterých uloží loginy studentů. Funkce nejprve otevře soubor s názvy loginů. Pokud se mu to nepodaří, vypíše chybovou hlášku a vrací NULL. Struktura je alokována dynamicky. Na začátku se alokuje na výchozí velikost 32 struktur STUDENT. Poté se ze souboru postupně načítají loginy, které se ukládají do struktury dokud se nenarazí na konec souboru. Pokud je počet loginů vyšší, než aktuálně alokovaná paměť, zvýší se alokace o dalších 32 položek. Po načtení se pomocí reference na číslo počtu loginů uloží počet načtených loginů a vrátí se ukazatel na strukturu s loginy.

3.9.4 Funkce „por_referencni“

Vstupem je pole polí čísel a počet studentů. Funkce postupně porovná všechna pole čísel studentů s referenčním řešením (nulté pole polí čísel) a vrátí přísnost (nejmenší počet rozdílů od referenčního řešení - 1). Využívá přitom funkce „porovnani_poli“. Pokud je však nejmenší počet rozdílů nula, vrací nulu.

3.9.5 Funkce „smazani_tabulek“

Vstupem funkce je pole hashovacích tabulek. Funkce na jednotlivé tabulky zavolá funkci „htSmazaniVsechRadku“ a poté uvolní paměť, kterou tabulky zabíraly.

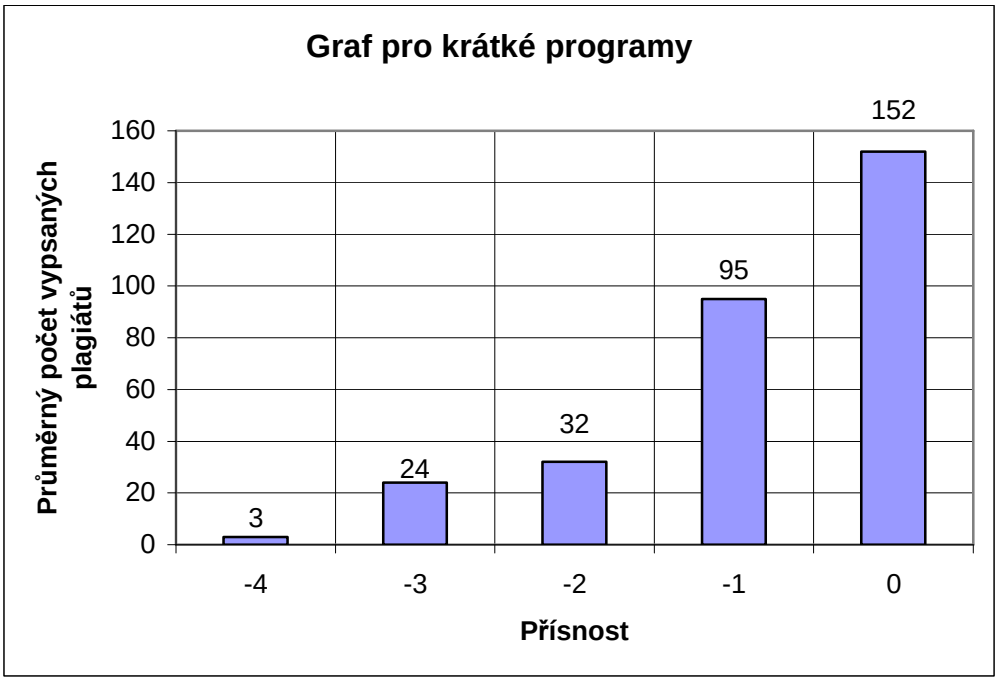
4 Výsledky, zhodnocení

4.1 Test aplikace na domácích úlohách v předmětu Algoritmy

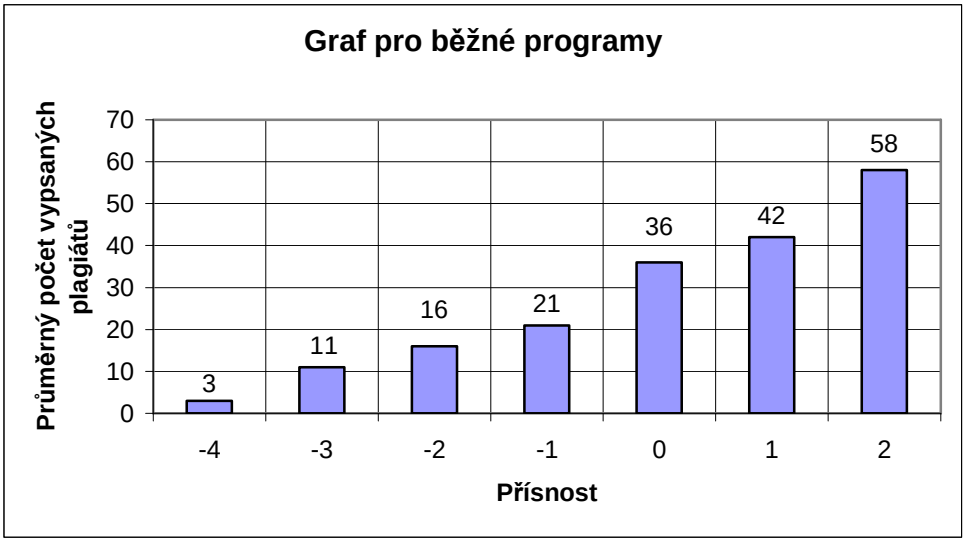
Aplikace byla testována na kontrolu obou domácích úloh v předmětu Algoritmy [8] v akademickém roce 2005/2006 a 2006/2007. Pomocí úloh z akademického roku 2005/2006 byla aplikace testována, vylepšována a doplňována tak, aby rozpoznala co nejvíc plagiátů a zároveň, aby se originály nevyhodnotily jako plagiáty.

Domácí úlohy z akademického roku 2006/2007 byly touto aplikací již kontrolovány. Od doby této kontroly došlo ještě k mnoha vylepšením této aplikace. U některých zdrojových kódů, kde bylo v každé funkci doplnit průměrně dva řádky kódu se vypisovalo příliš mnoho plagiátů, neboť tam není moc možností. Proto se zavedly i menší přísnosti aplikace (viz kapitola 2.6).

4.2 Statistiky



U domácích úloh, ve kterých se v jednotlivých funkcí doplňují průměrně dva řádky kódu (například funkce pro zásobník, frontu či vyhledávací tabulku v neseřazeném poli) je vhodné volit přísnost -3 (neberou se v úvahu komentáře, nadbytečné bílé znaky apod.) popřípadě -2 (zpracování navíc identifikátorů a čísel).



U ostatních domácích úloh je vhodné volit přísnosti -1 a 0. V případě programů, kde je více možností tvůrčí činnosti, lze volit i vyšší přísnosti.

Pro zjednodušení zvolení vhodné přísnosti je možnost testovat zdrojové kódy pro zvolený interval přísností. Například při zvolení intervalu od přísnosti -4 po 2 se vypíší plagiáty postupně pro všechny tyto přísnosti. Díky tomu se i zjednoduší následná ruční kontrola jednotlivých zdrojových kódů.

4.3 Přehled zápisu různých syntaxí konstrukcí jazyka C

Jak již bylo zmíněno v první kapitole, v jazyce lze mnoho stejných konstrukcí psát různou syntaxí. V této kapitole budou shrnuty všechny syntaxe, které aplikace bere v úvahu, při odhalování plagiátů.

Bílé znaky, blokové a řádkové komentáře

Všechny nadbytečné bílé znaky, řádkové a blokové komentáře se ignorují.

Identifikátory

Na názvu identifikátoru nezáleží, tzn. nezáleží na tom, jak se pojmenují proměnné, stejně tak i jména funkcí.

Inkrementace, dekrementace

x++; <==> x+=1; <==> x=x+1;
x--; <==> x-=1; <==> x=x-1;

Konstanty

a='A'; <==> a=65; <==> a=0x41;

Čísla

0.5 <==> .5
0.005 <==> 5E-3 <==> 5e-3
0x5a <==> 90 <==> 0132

Struktury

1.
```
struct {
 int vyska;
} pavel, honza, karel;
```

2.
```
struct miry {
 int vyska;
} pavel, honza, karel;
```

3.
```
struct miry {
 int vyska;
};
struct miry pavel;
struct miry honza, karel;
```

Všechny tři uvedené definice struktury jsou vyhodnoceny stejně. Tato syntaxe funguje to i pro uniony a výčtové typy.

Přístup k prvku ve struktuře:

(*karel).vyska <==> karel->vyska

Proměnné

int x; int y; x=10; y=1; <==> int x=10, y=1;
int x, y; x=10; y=1; <==> int x, y=1; x=10;

Datové typy

unsigned int <==> unsigned
long long int <==> long long
long int <==> long
short int <==> short
Pro všechny datové typy je v poli prvků vyhrazen jeden společný prvek, takže nezáleží, zda jeden student použije int a druhý long apod.

Podmínky

if ((a>0) && (b>0)) <==> if (!((a=<0) || (b=<0)))
if (a>0) prikaz1(); else prikaz2(); <==> if (a=<0) prikaz2(); else prikaz1();
if (a>0) <==> if (a>=1)

4.4 Možné rozšíření do budoucna

Do budoucna je možno rozšířit aplikaci o tyto vlastnosti:

- důkladná analýza komentářů a hledání v nich stejných pravopisných chyb atd.
- podpora modulárních programů
- další zvýšení inteligence rozpoznávání konstrukcí, odstranění mrtvého kódu

Závěr

Cílem bakalářské práce bylo seznámit se se všemi konstrukcemi programovacího jazyka C a poté navrhnout a implementovat aplikaci, která dokáže odhalit dva potenciálně stejné programy. Aplikace byla testována celkem na dvanácti zdrojových kódech domácích úloh v předmětu Algoritmy z předchozích dvou let. Pro jednoduché domácí úlohy se v základní verzi programu vypisovalo i na nulovou přísnost velké množství plagiátů, neboť studenti měli v každé funkci doplnit průměrně jen dva řádky kódu. Z toho důvodu se zavedly menší přísnosti této aplikace a dosáhlo se tak přiměřeného počtu plagiátů. Přísnosti jsou zvoleny tak, aby mezi nimi byl vždy přiměřený nárůst odhalených plagiátů.

Protože pro každou domácí úlohu je vhodná jiná přísnost, lze zvolit interval přísností, pro které se má testovat. Tím se vypíšou plagiáty postupně pro všechny přísnosti. Díky tomu se pak i usnadní následná ruční kontrola.

Rychlost programu je velká. Kontrola 500 zdrojových kódů na několik přísností je provedena v řádu několika vteřin. I přes velkou rychlost kontroly dosahuje dobrých výsledků při odhalování plagiátů.

Aplikace byla testována na 32-bitových operačních systémech Microsoft Windows XP, Red Hat 9 a FreeBSD.

Vzhledem k předloženým výsledkům můžeme konstatovat, že cíl bakalářské práce byl splněn.

Literatura

[1] Herout, P. *Učebnice jazyka C - 1. díl.* IV. přepracované vydání. KOOP, České Budějovice, 2004. ISBN 80-7232-220-6.

[2] Herout, P. *Učebnice jazyka C - 2. díl.* II. přepracované vydání. KOOP, České Budějovice, 2004. ISBN 80-7232-221-4.

[3] Wikipedie: Otevřená encyklopedie. *C (programovací jazyk)* [online]. [cit. 10.04.2007]. Dostupné z URL: <*http://cs.wikipedia.org/wiki/Jazyk_C*>.

[4] Šaloun, P. *Programování v jazyce C* [online]. [cit. 10.04.2007]. Dostupné z URL: <*http://docs.linux.cz/programming/c/c_saloun/*>.

[5] Růžička, D. *Učíme se jazyk C* [online]. [cit. 29.04.2007]. Dostupné z URL: <*http://www.builder.cz/serial3.html*>.

[6] Kernighan, B. W. – Ritchie, D. M. *Programovací jazyk C*. Computer Press, a.s., Brno, 2006. ISBN 80-251-0897-X.

[7] Kadlec, V. *Učíme se programovat v jazyce C*. Computer Press, Praha, 2002. ISBN 80-7226-715-9.

[8] Honzík, J. M. *Algoritmy*. [online]. [cit. 03.05.2007]. Dostupné z URL: < *http://www.fit.vutbr.cz/study/course-l.php?id=57*>.

Seznam příloh

Manuál k aplikaci

Aplikace se spouští s těmito parametry:

- cesta, kde se nachází kontrolované adresáře
- název kontrolovaného souboru
- název souboru, ve kterém se nachází názvy kontrolovaných adresářů (loginy studentů)
- přísnost
- počet znaků komentářů (nepovinný parametr)

Soubor, ve kterém se nachází názvy kontrolovaných adresářů se vytváří:

1. pro Windows příkazem: dir *cesta**.* /b type > *nazevsouboru*
2. pro UNIX příkazem: ls *cesta* > *nazevsouboru*

cesta - místo, kde se nachází kontrolované adresáře

Příklad pro Windows: dir du2*.* /b type > loginy.txt
Příklad pro UNIX: ls ./du2 > loginy.txt

Příklady jsou také umístěny v dávkovém souboru „vytvor.bat“ (pro Windows) a skriptu „vytvor.sh“ (pro UNIX). Oba soubory se nachází na přiloženém CD.

Přísnost se uvádí v od hodnoty -4. Význam jednotlivých přísností je uveden v technické zprávě v kapitole 2.6. Při zadání menší přísnosti se automaticky bere přísnost -4. V případě zadání přísnosti větší, než 99, provádí se velká přísnost a to s takovým počtem rozdílů, jaký je nejmenší počet rozdílů od referenčního řešení. Referenční řešení musí mít název adresáře „x“ a musí být v souboru s názvy kontrolovaných adresářů umístěn na prvním místě. Lze zadat i interval přísnosti. V tom případě bude aplikace testovat na všechny přísnosti v zadaném intervalu.
Příklad zadání intervalu přísnosti:

- 3:5 (testovat se bude pro přísnosti 3, 4 a 5)
- -4:1 (testovat se bude pro přísnosti -4, -3, -2, -1, 0 a 1)

Počet znaků komentářů značí, kolik znaků musí být maximálně navíc od kostry programu, aby byl zdrojový kód považován za nekomentovaný a byl tedy vypsán. Pokud se zadá záporné číslo, nebude se rozdíl počtu znaků vypisovat. Pokud součástí testování není kostra programu, vypíše se celkový počet znaků komentářů všech studentů. I zde platí, že pokud se dá počet znaků komentářů záporné číslo, nic se nevypisuje. V případě, že tento parametr vůbec nezadáme, bere se tento parametr jako nula.

Výpisy aplikace se provádí na obrazovku. Avšak výstup můžeme přesměrovat do souboru zadáním za parametry „> *nazev*“, kde *nazev* je název souboru, do kterého se mají výpisy aplikace uložit.

Příklady spuštění aplikace:

Windows:

```
plagiat du2/ c401.c loginy.txt 2 100 > c401.txt
plagiat du1/ c201.c loginy.txt -4:2 -1 > c201.txt
```

UNIX:

```
./plagiat ./du2/ c401.c loginy.txt 2 100 > c401.txt
./plagiat ./du1/ c201.c loginy.txt -4:2 -1 > c201.txt
```

Příklady spuštění aplikace jsou také umístěny v souborech test.bat (pro Windows) a tesh.sh (pro UNIX), které se nachází na přiloženém CD.