Algoritmická analýza úlohy

Centrálním požadavkem je vyhodnocování řetězcového výrazu, v jehož zápisu se nevyskytují syntaktické chyby. Celý výraz se skládá z volání funkcí, jejichž parametry mohou být opět výrazem. Rekurzivní povaha úlohy je patrná na první pohled, navíc si musíme uvědomit, že zápis výrazu v řádku není ničím jiným než zápisem kořenového stromu. Listy stromu odpovídají hodnotám řetězcových nebo číselných konstant nebo hodnotám daných proměnných. Vnitřní uzly stromu představují dané funkce I, D, L, přičemž hodnotu parametrů těchto funkcí lze standardně získat rekurzivním vyhodnocením jednotlivých podstromů příslušného vnitřního uzlu. Kořen stromu odpovídá nejvíce vnější operaci v daném výrazu.

Zpracování takovéhoto stromu průchodem v pořadí Postorder je nabíledni. Pro každý typ listu (řetězec, číslo, proměnná) potřebujeme separátní funkci, která ze vstupního výrazu přečte podřetězec specifikující hodnotu tohoto listu a vrátí jeho hodnotu. Pro všechny vnitřní uzly potřebujeme jedinou další rekurzivní funkci, která čte vstupní řetězec, aby určila, kterou z operací I, D, L uzel představuje. Potom funkce zavolá několikrát sama sebe, aby zjistila hodnoty svých parametrů, to jest, aby vyhodnotila podstromy daného uzlu, a nakonec spočte výsledek operace a předá jej volající funkci to jest nadřízenému uzlu. Každá instance funkce zpracovávající uzel musí navíc informovat svou volající instanci zpracovávající nadřízený uzel o tom, kde v daném vstupním výrazu končí úsek zpracovávaný v aktuálním uzlu, aby volající instance věděla, kde má pokračovat ve čtení celého vstupního výrazu.

Další drobné náležitosti řešení, jako je obsloužení přiřazovacího příkazu, větší počet řádků s příkazy atd., představují jednoduché akce, které jsou uvedeny dále v Programátorském komentáři k úloze resp. v ukázkovém kódu.

Programátorský komentář k úloze

Jak vyplývá z algoritmické analýzy, k implementaci není zapotřebí žádných mimořádných prostředků, díky rekurzivitě není třeba zavádět žádné (natožpak nějaké speciální) datové struktry, jedinou globální strukturou bude pole 26 řetězců představující jednotlivé řetězcové proměnné.

Základní úloha

Abychom si řešení úlohy co nejvíce usnadnili a mohli ji začít co nejdříve ladit nad smysluplnými daty, definujeme si tzv. základní úlohu, která bude speciálním případem dané úlohy, ale bude obsahovat skoro všechny důležité obraty. Dopsání kódu implementujícího původní úlohu pak již bude z velké části mechanickou záležitostí. V přiložené ukázce je to program `ExpressionBasic`.
V základní úloze

1. program obsahuje jen jediný řádek a ten má na pravé straně řetězcový výraz,
2. výraz na pravé straně obsahuje jen řetězcové a číselné konstanty, z funkcí obsahuje jen funkci Insert,
3. program neobsahuje mezery.

Funkce pro základní úlohu

Celou implementaci základní úlohy rozdělíme do čtyř funkcí. Menší počet funkcí by program jen znepřehledil, větší počet by jej naopak udělal příliš vyumělkovaným.

Funkce `readStringConstant(String inputSg, int index)`
přečte ze vstupního řetězce `inputSg` počínaje na pozici `index` řetězcovou konstantu a vrátí její hodnotu. Vstupním řetězcem bude po celou dobu běhu naší implementace jeden řádek přečtený ze vstupu.

Funkce `readInetgerConstant(String inputSg, int index)`
přečte ze vstupního řetězce `inputSg` počínaje na pozici `index` celočíselnou konstantu a vrátí její hodnotu.

Funkce `functionInsert(String sgWhere, int position, String sgWhat)`
implementuje funkci Insert ze zadání úlohy. V programu ji budeme volat v okamžiku, kdy budou známy hodnoty všech jejích parametrů. Tato funkce nebude nic číst ze vstupního řetězce.

Funkce `evalExpression(String inputSg, int index)`
je centrální pro naši úlohu. Tato funkce přečte ze vstupního řetězce `inputSg` celý jeden numerický nebo řetězcový výraz začínající na pozici `index` a vrátí jeho hodnotu. Hodnoty parametrů tohoto výrazu zjistí velmi snadno rekurzivním voláním sama sebe.

Návratové hodnoty funkcí
Funkce, které čtou a vyhodnocují nějaký výraz nebo konstantu uloženou v nějaké části vstupního řetězce, musí pro správné fungování celku vracet dvě hodnoty. Jednak je to sama číselná nebo řetězcová hodnota přečteného výrazu/konstanty a za druhé to musí být index prvního symbolu, který následuje za právě přečteným výrazem/konstantou, aby volající funkce věděla, kde má se čtením vstupního řetězce pokračovat.
V C/C++ a Javě není možné, aby funkce vracela více hodnot než jednu, musíme si proto sami vytvořit mechanizmus, který to provede. Řešení pomocí globálních proměnných je díky rekurzivnímu kódu nepoužitelné, zbývá definovat vlastní strukturu (C/C++) nebo třídu (Java) obsahující prostor pro obě hodnoty a pak funkce budou vracet referenci na instanci této struktury/třídy a odtud si volající funkce potřebné hodnoty vyzvedne. V ukázkové implementaci k tomu slouží třída `TermValue`.

Rozšíření základní úlohy
Vyzbrojeni z předchozích odstavců se již můžeme pustit do psaní (nebo čtení) kódu základní úlohy. Struktura kódu je natolik přehledná, že je jasné, že na doplnění do původní úlohy je nutno udělat jen následující:

1. dopsat funkce pro operace Delete a Length analogicky podle funkce functionInsert, to je velmi přímočará činnost, na několik, řekněme nejvýše deset až patnáct, minut,
2. rozšířit příkaz switch ve funkci `evalExpression` pro funkce Delete a Lenth, zde se hodí především metoda CtrlC-CtrlV (viz kód). Pokud by se potom tělo funkce `evalExpression` příliš protáhlo, lze snadno obsah jednotlivých větví `case` umístit do samostatných funkcí,
3. umožnit zpracovávat více vstupních řádků, což zajistí jediný kratičký cyklus,
4. obsloužit proměnné a výskyt jejich identifikátorů ve výrazech, to zajistí jediné pole řetězců a několikařádková varianta `case` pro tento případ ve funkci `evalExpression`,
5. přeskakovat nevýznamné mezery, což zajistí jediná další nejvýše dvouřádková funkce skipBlanks volaná na správných místech ve funkci `evalExpression`,
6. strukturovat pěkně činnosti popsané v 1.— 5., aby program zůstal snadno čitelný.