УДК 004.43

Списки и строки в предикатном программировании

Шелехов В.И. (Институт систем информатики СО РАН,
Новосибирский государственный университет)

Определяются языковые средства и методы реализации строковых объектов. Строки реализуются как массивы литер в стиле языков C и C++. Для строковых объектов доступен весь аппарат работы со списками. В связи с этим в данной работе пересматривается язык списков. В работе представлена часть библиотеки для строковых объектов.

Ключевые слова: *Функциональное программирование, предикатное программирование, автоматное программирование, список, строки в языке C.*

1. Введение

Предикатная программа состоит из набора программ на языке P [2] (определений предикатов) следующего вида:

```
<имя программы>(<описания аргументов>: <описания результатов>)
    pre <предусловие>
    { <оператор> }
    post <постусловие>
```

В предикатном программировании запрещены такие языковые конструкции, как циклы и указатели, серьезно усложняющие программу. Вместо циклов используются рекурсивные функции, а вместо массивов и указателей – списки и деревья. Предикатная программа проще в сравнении с императивной программой, реализующей тот же алгоритм.

Эффективность предикатных программ достигается применением при трансляции следующих оптимизирующих трансформаций:

- замена хвостовой рекурсии циклом;
- подстановка тела программы на место ее вызова;
- склеивание переменных: замена всех вхождений одной переменной на другую переменную;
- кодирование алгебраических типов (списков, деревьев, …) с помощью массивов и указателей.

В результате получается программа на императивном расширении языка P, которая далее конвертируется на язык C++. Отметим, что язык императивного расширения недоступен пользователю.

В данной работе трансформации операций со списками и строками рассматриваются лишь частично. Цель работы – определить в рамках языка P язык списков и строк, обеспечивающий надежность и эффективность программы. Тем не менее, особенности кодирования списков и строк в императивном расширении языка P в существенной степени предопределяются предлагаемыми языковыми возможностями.

В языках программирования наблюдается пестрая картина в реализации строковых объектов. Во многих функциональных языках, например в Haskell, строковый тип string определен как список литер; в коде строковый объект реализуется односвязным списком. В большинстве императивных языков тип string не определен – возможности работы со строковыми объектами обычно представлены в виде библиотеки. В языках C и C++ строковые объекты представляются указателями на массивы литер, завершающиеся нулем. Кроме того, имеются библиотеки на языке C++ с другим способом представления строк.

Строковый тип string – стандартный тип в языке предикатного программирования P [2], определяемый как list(**char**). Применимы все операции, определенные для списков. Строковые объекты представляются как в языках C и C++. Нуль как ограничитель значения строкового типа не входит в само значение, однако используется в алгоритмах со строками. Фактически вместо проверки на nil реализуется проверка на нуль для очередной литеры. По этой причине библиотеки для списков неприменимы для строковых объектов.

В языке P основным представлением списка является массив. С учетом того, что все действия по заказу и освобождению памяти под списки полностью скрыты от программиста, эффективная реализация операций со списками является проблематичной. Для достижения эффективности язык списков расширен дополнительными возможностями. В частности, допускается задание максимального числа элементов списка.

Описание нового языка списков и методов его реализации дано во втором разделе настоящей работы. Операции с памятью для списков и другие особенности описываются во третьем разделе. Четвертый раздел определяет язык строк на базе определенного ранее аппарата списков. Описываются особенности, определяемые способом кодирования строк с использованием завершающего нуля. В пятом разделе представлена часть программ из библиотеки для строковых объектов. Эта библиотека, в отличие от библиотеки на C++, не является классом. В соответствии с объектно-ориентированной технологией, класс скрывает

от пользователя детали реализации, в частности механизм отведения и освобождения памяти для строковых объектов. В предикатном программировании предполагается другой стиль: программист полностью контролирует эффективность своей программы. В шестом разделе приведен пример программы по обработке текстов.

Работа выполнена при поддержке РФФИ, грант № 12-01-00686.

2. Списки

Тип «список» – встроенный алгебраический тип со следующим определением:

```
type list (type T) = union (
    nil,
    cons(T car, list(T) cdr)
);
```

Здесь T – тип элемента списка, nil и cons – конструкторы. Канонический способ работы со списками определяется оператором выбора:

```
switch (s) {
    case nil: <оператор1>
    case cons(c, y): <оператор2>
};
```

Вхождения переменных c и y являются определяющими, причем c – начальный элемент списка s, а y – «хвост» списка s. Оператор выбора эквивалентен следующему оператору:

```
if (nil?(s)) <оператор1> else {{ T c = s.car || list(T) y = s.cdr}; <оператор2>};
```

Вместо распознавателя nil?(s), истинного при пустом списке s, привычнее использовать условие s == nil. Отметим, однако, что отношение равенства не определено для двух произвольных списков.

Определим набор языковых конструкций для работы со списками.

<выражение типа список> ::=
    <терм типа список> | <конкатенация списков> |
    <список-агрегат> | <конструктор списка> | <терм типа элемента списка>

Список-агрегат – это агрегат, определяющий список перечислением его элементов. Например: [[a, b, c]]. Терм типа элемента списка в позиции выражения типа список интерпретируется как одноэлементный агрегат.

<терм типа список> ::=
    <переменная типа список> | <терм типа список> . cdr |
    prec (<выражение типа список>) | <вырезка списка>

Пусть s – список. Тогда s.cdr определяет хвост списка, т.е. список без начального элемента; prec(s) – список без последнего элемента.

<конкатенация списков> ::=
    <выражение типа список> + <терм типа список> |
    <выражение типа список> + <терм типа элемента списка>
<вырезка списка> ::=
    <терм типа список> [<выражение> .. <выражение>] |
    <терм типа список> [<выражение> ..]

Значением вырезки вида s[m..n] является список, начинающийся элементом с номером m и заканчивающийся элементом с номером n. Элементы списка нумеруются с нуля. Значением вырезки будет пустой список (значение nil), если m больше номера последнего элемента, либо если n < m. Значением вырезки вида s[m..] является список, начинающийся элементом с номером m и включающий все элементы до конца списка.

<конструктор списка> ::=
    nil |
    cons (<терм типа элемента списка> , <выражение типа список>) |
    <специальный конструктор списка>

Здесь nil и cons(c, y), где c – элемент списка и y – список, являются стандартными конструкторами в соответствии с определением типа list. Специальный конструктор списка определен в следующем разделе.

<терм типа элемента списка> ::=
    <переменная типа элемента списка> | <терм типа список> . car |
    last (<выражение типа список>) | <терм типа список> [<индекс>]

Пусть s – список. Тогда s.car – начальный элемент списка; last(s) – список без последнего элемента; s[m] – элемент списка s с номером m. Напомним, что элементы в списке нумеруются с нуля.

Имеются другие конструкции со списками. Пусть s – выражение типа список. Тогда len(s) – длина списка s, т.е. число элементов в списке; nil?(s) – распознаватель, истинный для пустого списка s; cons?(s) – распознаватель, истинный для непустого списка s. Вместо данных распознавателей могут использоваться отношения s == nil и s != nil.

Отметим, что все конструкции, за исключением вырезки вида s[m..], в точности соответствуют описанию языка P [2]. Новые по отношению к [2] конструкции определены в следующем разделе.

3. Реализация списков

Основным способом представлением списка является массив. В качестве возможных альтернатив рассматриваются другие способы в виде: односвязного списка,

двунаправленного списка, кольцевого списка. Представление в виде кольцевого буфера следует считать модификацией представления в виде массива.

Далее рассматривается лишь представление списка в виде массива.

В соответствии с формальной семантикой языка P вычисление нового значения списка как результата некоторой операции сопровождается выделением памяти для этого значения. Буквальная реализация этого положения оказалась бы весьма расточительной. Например, при исполнении оператора s = x + y + z предполагается отведение памяти для результатов выражений x + y и (x + y) + z, а также для нового значения переменной s. Если заранее подсчитать длину списка x + y + z и отвести достаточную память для переменной s, то исходный оператор заменяется последовательностью "s = x; s = s + y; s = s + z", исполнение которой не требует дополнительной памяти.

Для любых видов строковых выражений, определенных в предыдущем разделе, возможен такой способ реализации, при котором удается отложить отведения памяти до момента присваивания списковой переменной, находящейся в левой части оператора присваивания. Поэтому отведение памяти далее рассматривается по отношению к списковым переменным.

Оптимальной реализацией является использование одного экземпляра памяти для всех присваиваний одной списковой переменной. Такое возможно, если известно верхнее ограничение L числа элементов списка.

Для изображения типа списка допускается использование следующих типовых термов:

list(T)

list(T, L)

Здесь T – тип элемента списка, L – максимальная длина списка. Размер памяти, отводимой для переменной типа list(T, L), будет достаточным для размещения L элементов списка.

Допустим, отведенная для списковой переменной память есть массив A с индексами в диапазоне 0..N. Тогда значение списковой переменной можно представить вырезкой A[m..n], где $0 \le m \le n \le N$, однако в случае пустого списка $m > n$. В большинстве случаев $m = 0$ и свободное место в памяти остается слева в диапазоне m+1..N. Однако бывают случаи, когда свободную часть памяти надо оставить справа для того, чтобы реализовать присваивание вида s = y + s, как, например, в работе [4], где используется присваивание buf = stf + buf. Для формирования нестандартного размещения значения списка используется специальный конструктор.

<специальный конструктор списка> ::=
    consLeft (<выражение типа список> , <индекс>) |
    consRight (<выражение типа список>) |
    consRight (<выражение типа список> , <максимальная длина>)

Конструктор вида consL(y, m), где m – номер элемента, формирует представление списка y в массиве, сдвинутое на m элементов относительно начала массива. Конструктор вида consRight(y) формирует представление списка y в массиве прижатым вправо. Конструктор вида consRight(y, L) формирует представление списка y в массиве длины L прижатым вправо. При исполнении оператора s = consRight(y, L) отводится новая память для строковой переменной s; если до присваивания переменная s уже имела некоторое значение, то транслятор с языка P должен обеспечить возврат старой памяти переменной s.

В языке P нет операторов отведения и освобождение памяти для списковых переменных. Вставка в код соответствующих действий реализуется транслятором. Отведение памяти реализуется при первом присваивании переменной. Размер памяти определяется по значению правой части оператора присваивания, если он явно не указан описанием типа.

Для значения списковой переменной не допускается выход значения за границы памяти, отведенной для переменной. Соответствующий контроль возлагается на программиста. Для этой цели предусмотрены следующие конструкции.

max_len(s)
store(s)
left_store(s)
resize(s, n)

Здесь s – переменная типа список. Значением функции max_len(s) является максимальное число элементов списка, которое можно разместить в массиве для переменной s. Функция store(s) определяет число элементов, которое можно разместить справа от значения s в свободной части памяти. Функция left_store(s) определяет число элементов, которое можно разместить слева от значения s в памяти. Оператор resize(s, n) отводит новую память размера n элементов, переписывает туда значение переменной s, освобождая старую память.

В дополнение к основному режиму, в котором программист полностью контролирует распределение памяти для списковых переменных, следует также предусмотреть режим, задаваемый прагмой, в котором контроль выхода за границы и заказ памяти большего размера реализуется автоматически.

Будем различать следующие виды присваиваний списковым переменным: создание (копирование), модификацию и сканирование. Данная классификация распространяется

также на аргументы вызова типа список, поскольку они рассматриваются как операторы присваивания соответствующим формальным параметрам.

Далее будем использовать следующие обозначения: s и y – переменные типа список, e и d – выражения типа список, m и n – переменные типа **nat**.

Присваивание вида копирования для оператора s = e реализуется копированием списка, вычисленного выражением e, в память для значения переменной s. Для оператора s = e + d в массив для хранения переменной s копируется значение e и вслед за ним копируется значение d.

Присваивание вида модификации[1] реализует изменение значения, размещаемого памяти для переменной s. Итоговое значение помещается в тот же участок памяти. Оператор s = s + e копирует список (значение выражения e) вслед за значением списка s в памяти. Оператор s = e + s копирует список (значение выражения e) перед значением списка s в памяти. Оператор s = s.car реализует отсечение хвоста списка. Операторы s = s[m..n] и s = s[m..] реализуют сужение значения списка к вырезке исходного значения s. Вместо сдвига значения s в начало массива проводится соответствующая корректировка позиции значения списка внутри памяти для списковой переменной. Операторы s = s.cdr и s = prec(s) также реализуются коррекцией позиции в памяти.

Списковая переменная, все действия с которой реализуют лишь анализ списка с продвижением по нему без его модификации[2] в памяти, называется *переменной сканирования.* Для переменной сканирования s присваивание вида s = y реализуется не копированием значения y, а создание *объекта сканирования*, ассоциированного с переменной y, и присваиванию его переменой s. Операторами сканирования являются s = s.cdr, s = prec(s), s = s[m..n] и s = s[m..]. Их отличие от соответствующих операторов присваивания в режиме модификации в том, что они не модифицируют переменной y. Операторы сканирования являются аналогами итераторов в императивных языках.

4. Строковый тип

Строковый тип string является предопределенным в языке предикатного программирования P [2]. Его определение имеет вид:

```
type string = list(char);
```

[1] Модификация переменных возможна как в исходной предикатной программе, так и в результате склеивания переменных после проведения трансформации склеивания переменных.

[2] Точнее, допускается модификация отдельных элементов списка без изменения их состава.

Набор средств, определенных в разделах 3 и 4 для списков, применим также для строк с некоторыми ограничениями. В дополнении к этому в языке P определены строковые константы.

Основным представлением строкового объекта является массив литер, завершающийся нулем, причем нуль не входит в значение строки. Иначе говоря, для строкового типа фактически действует следующее определение:

```
type string = subtype(list(char) s: s != nil & last(s) == 0);
```

[3]

Допускаются также другие представления строковых объектов; например, см. раздел 6. Однако в данном разделе рассматривается только основное представление.

Проверка строки s на пустоту реализуется оператором s.car == 0, а не s == nil. В принципе, возможна реализация, в которой конструктор nil кодируется значением из единственного нулевого элемента, однако это такое решение приведет к потере эффективности. В итоге, типы list и string несовместимы: со строковым объектом нельзя работать как со списком, в частности, нельзя подставлять строковый объект параметром типа list. Как следствие, библиотеки для списков неприменимы для строковых объектов.

Из-за специфики основного представления механизм реализации списков, описанный в разд. 3, лишь частично переносится на реализацию строковых объектов.

Следует сохранить возможность указания размера памяти для строковых объектов. В качестве альтернативного способа изображения строкового типа предлагается использовать string(L): память для значения типа string(L) вмещает ровно L литер. Полезны также конструкции, введенные для списков:

```
max_len(s)
store(s)
resize(s, n)
```

Исключается возможность сдвига вправо значения строки относительно начала памяти, т.е значение строки всегда размещается с начала памяти.

Целесообразно использовать два вида объектов сканирования: один – для сканирования с начала строки, второй – с конца. В первом случае объект сканирования может быть представлен указателем на начальный элемент строки, во втором – дополнительно требуется длина строки, при этом строка, представленная объектом сканирования, нулем не завершается.

---

[3] В операции сравнения предполагается неявное приведение last(s) к типу **int**.

Вводятся дополнительные конструкции для строковых объектов. Для определения числа элементов строки s вместо функции len(s) используется length(s). Конструктор nil, распознаватель nil?(s), а также отношения s == nil и s != nil не используются для строковых объектов. В качестве пустой строки используется конструктор empty, значением которого является строка из единственного нулевого элемента.

5. Некоторые программы из библиотеки для строковых объектов

Для объектов типа string не определено отношение равенства «==». Для лексикографического сравнения строк используется функция Compare с результатом 0 при совпадении строк, -1 – если первая строка меньше второй и 1 – если первая строка больше второй.

```
Compare(string s, t: int ) {
    char c = s.car, d = t.car;
    if (c == d) {
       if (c == 0) return 0 else return Compare(s.cdr, t.cdr)
    } else return c - d
}
```

Программа SubString заменяет строку s ее вырезкой длины n начиная с элемента по номеру p, т.е. вырезкой s[p..p + n]. При этом итоговая вырезка не может выходить за границу исходной строки s.

```
SubString(string s, nat p, n: string s'){
    if (p >= length(s)) s' = empty
    else s' = s[p..min(p + n, length(s) - 1)];
}
```

При p≠0 итоговое значение s′ получается сдвигом исходного значения s в памяти. Не следует использовать данную программу в случае, когда значение переменной s′ далее не модифицируется. Предпочтительней использовать операции вырезки, которая будет реализована через объект сканирования.

Программа Insert вставляет строку t внутрь строки s начиная с позиции p.

```
Insert(string s, t, nat p: string s'){
    if (p > length(s) or t == empty) return;
    nat L = length(s) + length(t);
    if (L > max_len(s)) resize(s, L);
    if (p = 0) { s' = t + s; return };
    s' = s[0..p-1] + t + s[p..]
}
```

Эффективная реализация оператора s′ = s[0..p-1] + t + s[p..] в императивном расширении обеспечивается парой вызовов программ из внутренней библиотеки:

```
right(s, p, length(s), p + length(t));
copy(s, t, p)
```

Вызов right сдвигает вырезку s[p..] вправо (вместе с завершающим нулем) на length(t) позиций. Вызов copy переписывает значение строки t в строку s начиная с позиции p.

Программа Replace заменяет часть строки, соответствующей вырезке s[p..p + n], на строку t. Если p + n выходит за границы строки s, заменяемая вырезка ограничена концом строки.

```
Replace(string s, t, nat p, n: string s′){
    if (p > length(s)) return;
    if (p = length(s)) { s′ = s + t; return };
    nat m = (p + n >= length(s))? length(s) – 1 : p + n;
    nat L = length(s) + length(t) – m + p – 1;
    if (L > max_len(s)) resize(s, L);
    s′ = s[0..p-1] + t + s[m+1..]
}
```

Функция StartsWith проверяет, является ли строка t начальной частью строки s.

```
StartsWith(string s, t: bool ) {
    if (t.car == 0) return true;
    if (s.car != t.car) return false;
    return StartsWith(s.cdr, t.cdr)
} post ∃ string u. s = t + u;
```

Функция EndsWith проверяет, является ли строка t конечной частью строки s.

```
EndsWith(string s, t: bool ) {
    return EndsWith(string s, t, length(s), length(t));
} post ∃ string u. s = u + t;

EndsWith(string s, t, nat js, jt: bool ) {
    if (s[js] != t[jt]) return false;
    if (jt == 0) return true;
    if (js == 0) return false;
    return StartsWith(s, t, js - 1, jt - 1)
};
```

Гиперфункция Index определяет первую позицию элемента c в строке s начиная с позиции p. Результат – позиция q. При отсутствии элемента c реализуется вторая ветвь #noel.

```
Index(string s, char c, nat p: nat q #yes : #noel){
    if (p >= length(s)) #noel;
    Index1(s, c, p: q #yes : #noel)
}

Index1(string s, char c, nat q: nat q' #yes: #noel){
    if (s[q] == c) {q' = q; #yes };
    if (s[q] == 0) #noel;
    Index1(s, c, q+1: q' #yes : #noel)
}
```

Гиперфункция RIndex определяет последнюю позицию элемента c в строке s не превышающую позиции p. Результат – позиция q. При отсутствии элемента c реализуется вторая ветвь гиперфункции #noel.

```
RIndex(string s, char c, nat p: nat q #yes: #noel){
    if (p>= length(s)) p = length(s) – 1;
    RIndex1(s, c, p: q #yes: #noel)
}

RIndex1(string s, char c, nat q: nat q' #yes : #noel){
    if (s[q] == c) {q' = q; #yes };
    if (q == 0) #noel;
    RIndex1(s, c, q – 1: q' #yes : #noel)
}
```

Гиперфункция Index определяет позицию первого вхождения строки t в строке s начиная с позиции p. Результат – позиция q. При отсутствии вхождений, а также в случае пустой строки t реализуется вторая ветвь гиперфункции #nostr.

```
Index(string s, t, nat p: nat q #yes: #nostr){
    if (t == empty) #nostr;
    Index1(s, t, p, length(s) – length(t) : q #yes : #nostr)
}
Index1(string s, t, nat p, Lts: nat q #yes : #nostr){
    if (p > Lts) #nostr;
    Index2(s, t, p, 0 : q #yes : )
    Index1(s, t, p+1, Lts: q #yes : #nostr)
}

Index2(string s, t, nat p, j: nat q #yes : #no){
    if (t[j] == 0) {q = p; #yes};
    if (t[j] != s[p+j]) #no;
    Index2(s, t, p, j+1: q #yes : #no)
}
```

После трансформаций замены рекурсии циклом и подстановки тел программ на место вызовов программа Index приводится к следующему виду:

```
Index(string s, t, nat p: nat q #yes: #nostr){
    if (t == empty) #nostr;
    nat Lts = length(s) – length(t);
    for (; ; p = p+1) {
        if (p > Lts) #nostr;
        for (j=0; ; j = j+1) {
            if (t[j] == 0) {q = p; #yes};
            if (t[j] != s[p+j]) break;
        }
    }
}
```

Гиперфункция RIndex определяет позицию последнего вхождения строки t в строке s не превышающую позиции p. Результат – позиция q. При отсутствии вхождений, а также в случае пустой строки t реализуется вторая ветвь гиперфункции #nostr.

```
RIndex(string s, t, nat p: nat q #yes : #nostr){
    if (t == empty) #nostr;
    if (p>= length(s)) p = length(s) – 1;
    RIndex1(s, t, p, length(t) – 1: q #yes : #nostr)
}

RIndex1(string s, t, nat p, Lt1: nat q #yes : #nostr){
    if (Lt1 > p) #nostr;
    RIndex2(s, t, p, Lt1 : q #yes : )
    RIndex1(s, t, p – 1, Lt1: q #yes: #nostr)
}

RIndex2(string s, t, nat q, j: nat q' #yes : #no){
    if (t[j] != s[q]) #no;
    if (j == 0) {q' = q; #yes};
    RIndex2(s, t, q – 1, j – 1: q' #yes : #no)
}
```

Гиперфункция FirstOf определяет позицию первого вхождения любого элемента строки t в строке s начиная с позиции p. Результат – позиция q. При отсутствии вхождений, а также в случае пустой строки t реализуется вторая ветвь гиперфункции #noel.

```
FirstOf(string s, t, nat p: nat q #yes: #noel) {
    if (t == empty) #noel;
    FirstOf1(s, t, p, length(s) : q #yes : #noel)
}
```

```
FirstOf1(string s, t, nat p, Ls: nat q #yes : #nostr){
    if (p >= Ls) #nostr;
    FirstOf2(s, t, p, 0 : q #yes : )
    FirstOf1(s, t, p+1, Ls: q #yes : #nostr)
}

FirstOf2(string s, t, nat p, j: nat q #yes : #no){
    if (t[j] = s[p]) {q = p; #yes};
    if (t[j] == 0) #no;
    FirstOf2(s, t, p, j+1: q #yes : #no)
}
```

6. Программа выделения первого слова

В данном разделе рассматривается программа обработки текста, представленная на сайте [1] в качестве иллюстративного примера при описании методов автоматного программирования А.А. Шалыто [3].

Требуется написать программу, читающую из потока стандартного ввода текст, состоящий из строк, и для каждой строки печатающую первое слово этой строки и перевод строки. Слова разделяются пробелами. Знаки препинания отсутствуют. Пробелы могут находиться в начале строки. Конец строки определяется литерой '\n', конец текста – литерой EOF. Литера конца текста обязательно присутствует в тексте.

В работе [4] определены два класса программ: программы-функции и программы-процессы. Программа принадлежит классу *программ-функций*, если она не взаимодействует с внешним окружением. Точнее, если возможно перестроить программу таким образом, чтобы все операторы ввода данных находились в начале программы, а весь вывод собран в конце программы. В противном случае программа принадлежит классу программ-процессов и адекватно представима в виде автоматной программы [4]. Применение автоматного программирования для программ-функций противопоказано. Автоматные программы по своей структуре существенно сложнее программ-функций.

Представленная программа обработки текстов относится к классу программ-функций. Посимвольно вводимый входной текст можно предварительно собрать в строковой переменной in. Единственная особенность – в качестве конца строкового значения используется литера EOF. Выходной текст можно накапливать в строковой переменной out и распечатать по завершению программы.

Сначала воспроизведем эквивалентную предикатную программу для исходной программы на сайте [1].

```
char EOF; // литера конца текста
char SP = ' '; // литера «пробел»
FirstWords(string in : string out) { FirstWords1(in, nil : out); }
```

Программа FirstWords сводится к более общей программе FirstWords1. Второй параметр – начальное значение переменной out.

```
FirstWords1(string in, string out : string out') {
    skipSpaces(in.cdr, in.car : string in1, char c);
    getWord(in1, c, out : string in2, char c2, string out1);
    string out2 = out1 + '\n';
    skipLine(in2, c2 : string in3, char c3);
    if (c3 != EOF) FirstWords1(in3, out2 : out');
}
```

Программа skipSpaces реализует пропуск возможных пробелов в начале очередной строки. Ее результатами являются: остаток входного текста in1 и последний прочитанный символ c, отличный от пробела. Использование переменной c необходимо во избежание повторного чтения литер из входного текста. Программа getWord выделяет начальное слово очередной строки и записывает его в переменную out. Программа skipLine сканирует остаток строки до ее конца. Ниже приведем более привычную для императивных программистов эквивалентную версию программу FirstWords1 с использованием модифицируемых переменных.

```
FirstWords1(string in, string out* : ) {
    skipSpaces(in.cdr, in.car : in, char c);
    getWord(in, c, out : in, c, out);
    string out = out + '\n';
    skipLine(in, c : in, c);
    if (c != EOF) FirstWords1(in, out : out);
}
```

Ниже представлены программы skipSpaces, getWord и skipLine.

```
skipSpaces(string in*, char c* :)
{   if (c == SP) skipSpaces(in.cdr, in.car : in, c); }

getWord(string in*, char c*, string out* :)
{    if (c != SP & c != '\n' & c != EOF)
        getWord(in.cdr, in.car, out + c : n1, c, out);
}

skipLine(string in*, char c* : )
{  if (c != '\n' & c != EOF) skipLine(in.cdr, in.car : in, c); }
```

Для приведенной предикатной программы наряду со штатными трансформациями применяются следующий набор определяемых пользователем трансформаций:

```
/*#  c = in.car; in = in.cdr → input(c)
   forall char x. out = out + x → print(x)
   out = nil →
#*/
```

Данные трансформации заменяют операции со строками in и out операторами input(c), print(c) и print('\n'). В результате трансформаций получим в точности исходную императивную программу на сайте [1]. Данная программа имеет очевидные дефекты в эффективности: проверка очередного символа на совпадение с пробелом и символами конца строки и конца текста реализуется дважды. Наша задача – показать, что в рамках предикатного программирования можно устранить эти и другие дефекты и провести предельную оптимизацию любого алгоритма. Разумеется, в реальном программировании подобные дефекты в большинстве случаев не являются критичными, однако имеются приложения, где любые потери эффективности недопустимы.

Ниже представлена эффективная программа обработки текста, использующая аппарат гиперфункций [4]. В ней изменены программы getWord и skipLine.

```
FirstWords1(string in, string out* : ) {
      skipSpaces(in.cdr, in.car : in, char c);
      getWord(in, c, out : in, out #sp: in, out #lf: in, out #e);
sp:   skipLine(in : in #lf: #e);
e:    out = out + '\n'; return
lf:   FirstWords1(in, out + '\n' : out);
 }
```

Программа getWord – гиперфункция с тремя ветвями, реализуемыми, соответственно, при достижении пробела, конца строки и конца текста. Поскольку текущий символ c после вызова skipSpaces отличен от пробела, проверка на пробел в getWord проводится для следующего символа.

```
getWord(string in*, char c, string out* : #sp:  #lf:  #e)
{   if (c == '\n')  #lf;
    if (c == EOF) #e;
    out = out + c; c = in.car; in =in.cdr;
    if (c == SP)  #sp;
    getWord(in, c, out : in, c, out);
}
```

```
skipLine(string in*: #lf: #e)
{    char c = in.car; in =in.cdr;
     if (c == '\n') #lf;
     if (c == EOF) #e;
     skipLine(in : in #lf: #e)
}
```

Покажем построение эффективной императивной программ методом трансформаций для новой версии предикатной программы. На первом этапе хвостовая рекурсия заменяется циклом.

```
FirstWords1(string in, string out* : ) {
for (;;) {
     skipSpaces(in.cdr, in.car : in, char c);
     getWord(in, c, out : in, out #sp: in, out #lf: in, out #e);
sp:   skipLine(in.cdr, in.car : in #lf: in #e);
e:    out = out + '\n'; return
lf:   out = out + '\n';
}
}

skipSpaces(string in*, char c* :)
{ while (c == SP) { c = in.car; in =in.cdr }}

getWord(string in*, char c, string out* : #sp:  #lf:  #e)
{  for (;;) {
     if (c == '\n')  #lf;
     if (c == EOF) #e;
     out = out + c; c = in.car; in =in.cdr;
     if (c == SP)  #sp;
}
}
skipLine(string in*,: #lf: #e)
{    for (;;) {
        char c = in.car; in =in.cdr;
        if (c == '\n') #lf;
        if (c == EOF) #e;
     }
}
```

На втором этапе трансформаций реализуется постановка тел программ skipSpaces, getWord и skipLine на место их вызовов. Затем тело FirstWords подставляется на место вызова. Получим:

```
  FirstWords(string in : string out) {
out = nil;
for (;;) {
      char c;
      do { c = in.car; in =in.cdr } while (c == SP);
      for (;;) {
          if (c == '\n')  #lf;
          if (c == EOF) #e;
          out = out + c; c = in.car; in =in.cdr;
          if (c == SP)  #sp;
      }
 sp:  for (;;) {
          c = in.car; in =in.cdr;
          if (c == '\n') #lf;
          if (c == EOF) #e;
      }
 e:   out = out + '\n'; return
 lf:  out = out + '\n';
}
  }
```

На третьей стадии трансформаций реализуются замены:

$$c = in.car;\ in = in.cdr \rightarrow input(c)$$

$$out = out + x \rightarrow print(x)$$

В итоге получим окончательную программу:

```
  FirstWords() {
for (;;) {
      char c;
      do input(c) while (c == SP);
      for (;;) {
          if (c == '\n')  #lf;
          if (c == EOF) #e;
          print(c); input(c);
          if (c == SP)  #sp;
      }
 sp:  for (;;) {
          input(c);
          if (c == '\n') #lf;
          if (c == EOF) #e;
      }
 e:   print('\n'); return
 lf:  print('\n');
}
  }
```

Заключение

В данной работе проведена доработка языка предикатного программирования [2] для списков и строк. Новые языковые средства обеспечивают лучший контроль эффективности программы, что соответствует стилю парадигмы предикатного программирования. В частности, обеспечивается аккуратная работа с памятью без явных операторов отведения и освобождения памяти. Это, однако, потребует дополнительной нагрузки на транслятор и используемый в нем потоковый анализ.

Отметим, что в отличие от предыдущих проектов языка списков и строк, не фиксируется представление в коде списков и строк. Возможный способ кодирования приводится лишь для объектов сканирования строк. Однако очевидно, представления списков и строк существенно различаются

В пятом разделе значительная часть библиотеки для строковых объектов (файлы string.h и string.cpp) переписана на язык P с использованием новых возможностей, представленных в разделе 3. Стиль работы с памятью можно проследить на примере программ SubString, Insert и Replace. Если предикатный программист правильно оценивает размеры строковых объектов, то новой памяти не потребуется, тогда как соответствующие программы в string.cpp всегда отводят память с оглядкой на возможность наличия нескольких ссылок на исходную строку. Следует отметить, что программа, использующая string.cpp, будет довольно часто «трясти» (заказывать и освобождать) память. Такой стиль программирования неприемлем для встроенных систем.

Следует отметить, что библиотека (string.h и string.cpp) не штатная, а скорее специальная с хакерским стилем реализации. Штатной видимо является библиотека для языка C, недавно формально верифицированная в работе [5].

Список литературы

1. Автоматное_программирование. 2014 [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Автоматное_программирование (дата обращения: 10.12.2014).
2. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.12. Новосибирск, 2013. 28с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf. (дата обращения: 10.12.2014).
3. Поликарпова Н.И., Шалыто А.А. Автоматное программирование / СПб.: Питер. 2009. 176с. [Электронный ресурс]. URL: http://is.ifmo.ru/books/_book.pdf (дата обращения: 10.12.2014).

4. Шелехов В.И. Язык и технология автоматного программирования // Программная инженерия, №4, 2014. С. 3-15. [Электронный ресурс].
URL: http://persons.iis.nsk.su/files/persons/pages/automatProg.pdf (дата обращения: 10.12.2014).
5. Carvalho, Nuno, et al. Formal Verification of kLIBC with the WP Frama-C Plug-in // NASA Formal Methods. Springer International Publishing, 2014. P. 343-358.

UDK 004.43

Lists and strings in predicate programming

Vladimir I. Shelekhov (A.P. Ershov Institute of Informatics Systems, Novosibirsk State University)

The language and implementation of string objects are defined. Strings are coded via arrays of chars in the style the C and C++ languages. List constructs are accessible for strings. The list language has been revised to be proper for strings. The paper presents some part of the program library for strings.

Functional programming, predicate programming, automata-based programming, list, strings in the C++ language.