ОДНОСТЕКОВАЯ РЕАЛИЗАЦИЯ БЭКТРЕКИНГА ДЛЯ ЯЗЫКА ФОРТ

М. Л. Гасаненко

Санкт-Петербургский институт информатики и автоматизации РАН
199178, Санкт-Петербург, 14-я линия В.О., д.39
http://www.forth.org.ru/~mlg

УДК 681.3
М. Л. Гасаненко. **Одностековая реализация бэктрекинга для языка Форт** // Труды СПИИРАН. Вып. 1, т. 1. — СПб: СПИИРАН, 2002.

Аннотация. *Предложена методика реализации механизма откатов (бэктрекинга, backtracking) для языка Форт, отличающаяся использованием только одного стека (стека возвратов) и совместимостью с механизмом локальных переменных. Обсуждается возможность применения аналогичных методов для расширения бэктрекингом языка Си. Бэктрекинг позволяет ввести в язык еще один вид модульности — модули, отвечающие за перебор.* — Библ. 13 назв.

UDC 681.3
M. L. Gassanenko. **A Single-Stack Implementation of Backtracking for Forth** // SPIIRAS Proceedings. Issue 1, v. 1. — SPb: SPIIRAS, 2002.

Abstract. *An approach to implementation of backtracking in Forth is proposed that uses a single stack (the return stack) and is compatible with local variables. Applicability of analogous techniques to implementation of backtracking for C is discussed. Backtracking introduces one more sort of modularity into the language: modules responsible for iteration.* — Bibl. 13 items.

1. Введение

1.1. Бэктрекинг в Форте

В работе [1] описана реализация механизма откатов (бэктрекинга, backtracking), отличающаяся пренебрежимо малыми потерями скорости исполнения, а также простотой идеи и небольшим размером реализации. Путем нестандартного использования стека возвратов (вспомогательной структуры данных фортовского интерпретатора) удалось избежать необходимости написания нового интерпретатора и связанных с этим потерь быстродействия; видимо, это следует классифицировать как повторное использование существующего интерпретатора. Слово же "бэктрекинг" здесь следует понимать в смысле языка Пролог — это не только метод организации перебора вариантов, но и синтаксис языка, позволяющий считать, что при откате программа исполняется справа налево. Двухстековая архитектура языка Форт подробно описана в [2].

Если считать, что дополнительный стек (L-стек) уже реализован, реализация бэктрекинга укладывается в три строки кода:

```
: ENTER >R ;          ( a -- )    \ вызвать фрагмент кода по адресу a
: PRO  R> R> >L ENTER LDROP ; \ адрес продолжения — на L-стек
: CONT L> >R R@ ENTER R> >L ; \ вызов продолжения (успех)
```

Продолжением будем называть остаток кода вызвавшей процедуры, при вызове адрес продолжения помещается на вершину стека возвратов. Слово PRO (от "prologue") переносит адрес продолжения со стека возвратов на L-стек (впоследствии, при откате, убирает его оттуда). *Успех* — это вызов продолже-

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

ния, т.е. остатка кода вызвавшей процедуры. Слово CONT (от "continue") выполняет вызов продолжения, используя адрес на L-стеке (на время исполнения продолжения адрес снимается с L-стека). *Откат* — возврат из продолжения — выполняется командой EXIT. (Если процедура выполнила команду PRO, то EXIT вызывает откат, и наоборот, в отсутствие слова PRO откат интерпретируется как выход.)

Пример использования:

```
: от1до10 PRO 11 1 DO I CONT LOOP ;
: четные  PRO DUP 2 MOD 0= IF CONT ELSE DROP THEN ;
: test    от1до10 . ;
: test2   от1до10 четные . ;
```

Слово test распечатывает числа от 1 до 10, а слово test2 — те из них, которые делятся на два.

Преимущество бэктрекинга: *алгоритмы перебора оформляются в виде отдельных модулей, пригодных для повторного использования.*

Это не единственная реализация бэктрекинга, еще два подхода описаны в работах [3, 4, 5, 6]. Подход из [3, 4] отличается использованием исполнения данных и передачей вспомогательного флага, дополнительных стеков не требуется; подход [5, 6] отличается использованием стека данных для хранения истории, что блокирует стек данных — это следует считать серьезным недостатком. В [3, 4, 5, 6] бэктрекинг используется как средство решения конкретной задачи и методика реализации оказывается привязанной к этой задаче, в то время как мы используем бэктрекинг для введения в язык еще одного вида модульности.

1.2. Цель работы

Описанная в работе [1] реализация бэктрекинга обладает следующими двумя недостатками.

1) Требуется отдельный стек.

2) Несовместимость с традиционной реализацией локальных переменных.

На первый взгляд, ни то, ни другое серьезным недостатком не является. Реализация еще одного стека на Форте занимает 8 строк кода. Реализации бэктрекинга и локальных переменных можно переписать так, чтобы они были совместимы друг с другом.

На практике, однако, все оказывается по-другому: эти недостатки чреваты проблемами на стадии поддержки программного обеспечения.

Чаще всего встречаются задачи, которые прекрасно решаются и без бэктрекинга, для них оформление алгоритмов перебора в виде отдельных программных единиц бывает полезно, но не необходимо. В таких условиях необходимость поддержки еще одного стека (при наличии многозадачности — для каждой задачи) является серьезным аргументом против использования бэктрекинга. В итоге, на практике бэктрекинг используется в том подмножестве, которое не требует дополнительного стека (полное описание см. в [1, 7]).

С локальными переменными ситуация аналогичная: о целесообразности их применения можно спорить. Языком Форт, с его явным доступом к стеку, можно эффективно пользоваться только тогда, когда определения короткие, в среднем около 7 слов (не считая ":" и ";"). (Еще одним существенным условием

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

является осмысленность каждого такого модуля, то есть каждый маленький модуль должен выражать некоторую идею — ровно одну). Модуль такого размера можно легко протестировать в режиме интерпретации из командной строки, и в случае неправильного поведения ошибка скорее всего будет очевидна.

Для такого стиля программирования использование локальных переменных — скорее исключение, чем правило.

Реализация совместимых механизмов локальных переменных и бэктрекинга "в лоб" приводит к тому, что ради гипотетического случая их совместного использования реализации неадекватно усложняются, теряется быстродействие, а будущее повторное использование кода становится маловероятным. Реализация пакета, завязанная на особенности одной из множества возможных реализаций еще одного пакета — вещь сомнительная, так как при переносе может потребоваться изменение исходного кода пакета по не связанным с этим пакетом причинам.

Итак, цель работы — не заводя дополнительных стеков, реализовать бэктрекинг и локальные переменные так, чтобы они не мешали друг другу. Способ реализации каждого из этих механизмов не должен зависеть от наличия или отсутствия другого механизма.

1.3. Воздействие на поток управления через доступ к стеку возвратов

До появления формального описания влияния манипуляций со стеком возвратов на поток управления такой доступ к стеку возвратов считался "грязным трюком" и не вошел в стандарт 1994 года [8].

Из работы [9] и, в частности, теоремы об эквивалентности вызова вытекает следующее правило замены открытой подстановки вызовом при наличии доступа к стеку возвратов.

Стек интерпретации состоит из указателя интерпретации IP и стека возвратов. Верхнее значение стека интерпретации, IP, все время изменяется. При вызове фрагмента кода на стек интерпретации добавляется еще одно значение; при выходе из процедуры со стека интерпретации снимается верхнее значение. Чтобы создать процедуру, производящую некоторые изменения на стеке интерпретации, нужно написать код, производящий такие же изменения на стеке возвратов, и поместить его внутрь вспомогательной процедуры. (При вызове процедуры содержимое стека интерпретации становится содержимым стека возвратов, а при выходе наоборот.)

Замечание. Если используются слова доступа к указателю стека возвратов RP@ и RP!, надо иметь в виду, что на вершине стека возвратов будет находиться еще одно значение — адрес возврата из вспомогательной процедуры.

1.4. Формальное описание доступа к стеку возвратов

В работах [9, 10, 11] предложен формализм на основе эквивалентных преобразований, а в работе [12] интерпретация шитого кода и доступ к стеку возвратов описаны с на языке лямбда-исчисления.

Вкратце изложим систему обозначений из [9]:

"==" обозначает эквивалентность (функциональную)

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

";" обозначает EXIT (в Форте ";" обозначает EXIT и конец определения, т.е. имеет несколько другой смысл)
[r ... '] обозначает вызов процедуры (в момент достижения символа "[r" адрес кода, находящегося за "']", помещается на стек возвратов.)

Для слова X , определенного как : X Y Z ;

```
X == [r Y Z ; ']
```

и

```
A X B == A [r Y Z ; '] B
```

т.е. в отличие от фортовской нотации, наша допускает открытую подстановку.
'r[X] обозначает помещение адреса фрагмента кода X на стек возвратов. Справедливо

```
[r X '] Z == 'r[ Z ] X
```

R1(X) обозначает фрагмент кода, отличающийся от X только тем, что верхний элемент стека возвратов игнорируется; если X описывается диаграммой (R: i*x -- j*x), то R1(X) описывается диаграммой (R: i*x x1 -- j*x x1). Определение выглядит как

```
N >R R1( X ) == X N >R ,
```

так что

```
R1( EXIT ) == RDROP EXIT .
```

Справедлива теорема (об эквивалентности вызова), что

```
X == [r R1( X ) ; ']
```

(т.е. чтобы при переносе куска кода внутрь процедуры эффект не изменился, надо изменить кусок кода так, чтобы он не трогал адрес возврата.)

Понятно, что если R1(X) == X , то есть если X не осуществляет доступа к стеку возвратов, то никаких таких проблем нет и допустима открытая подстановка тела вызываемой процедуры в код вызывающей процедуры.

Заметим, что

R1(*получить-значение* RP!) == R> *получить-значение* RP! >R

и

R1(RP@ *сохранить-значение*) == R> RP@ *сохранить-значение* >R

2. Переход к одностековой реализации бэктрекинга

2.1. Проведем эквивалентное преобразование кода реализации

```
: ENTER >R ;
: PRO   R> R> >L ENTER LDROP ;
: CONT  L> >R R@ ENTER R> >L ;

ENTER == [r >R ; ']

PRO == [r R> R> >L ENTER LDROP ; '] ==
== [r R> R> >L [r >R ; '] LDROP ; '] ==
== [r R> R> >L 'r[ LDROP ; ] >R ; '] ==
== R> >L 'r[ LDROP ; ]
```

Примем следующее обозначение:

```
>L{LDROP}  ==  >L 'r[ LDROP ; ]
```

тогда

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

```
PRO == R> >L{LDROP} == [r R1( R> >L{LDROP} ) ; '] ==
== [r R> R> >L{LDROP} >R ; ']
```

т.е. PRO можно определить как

```
: PRO R> R> >L{LDROP} >R ;
```

Проведем аналогичное преобразование для CONT

```
CONT == [r L> >R R@ ENTER R> >L ; '] ==
== [r L> >R R@ [r >R ; '] R> >L ; '] ==
== [r L> >R R@ 'r[ R> >L ; ] >R ; ']
```

Примем следующее обозначение:

```
L>{LREST} == L> >R R@ 'r[ R> >L ; ] ==
== L> DUP >R 'r[ R> >L ; ]
```

тогда

```
CONT == [r L>{LREST} >R ; ']
```

т.е. CONT можно определить как

```
: CONT L>{LREST} >R ;
```

2.2. L-стек может существовать не физически, а логически

Интересно, что для того, чтобы реализовать слова >L{LDROP} и L>{LREST} , не нужно отводить память под еще один стек. Можно вполне обойтись определениями:

```
VARIABLE LP      : LP@ LP @ ;  : LP! LP ! ;
: >L{LDROP} ( x --> )( l: --> x ) (  <--  )( l: <-- x )
  \ На стеке возвратов: ( r: -- x lp-old [<-lp] ra )
  R> SWAP
    LP@ 2>R RP@ LP!
    BACK R> LP! RDROP TRACKING
  >R ;
: L>{LREST} ( --> x )( l: x --> ) (  <--  )( l: x <-- )
  R>
    LP@ >R
    BACK R> LP! TRACKING
    LP@ 2@ LP!
  SWAP >R ;
: L@ LP@ CELL+ @ ;  ( -- x )
```

А как быть с имеющимся формализмом?
С одной стороны, у нас

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

```
>L{LDROP}_1  ==  >L 'r[ LDROP ; ]
```

с другой,

```
>L{LDROP}_2  ==
== [r R> SWAP LP@ 2>R RP@ LP! 'r[ R> LP! RDROP ; '] >R ; ==
== LP@ 2>R RP@ LP! 'r[ R> LP! RDROP ; ']
```

В работе [10] дано формальное описание бэктрекинга в предположении, что он реализован через L-стек, существующий и физически, и логически. Терять возможность пользоваться наработанными результатами не хочется, однако задача формального описания только что предложенного подхода к реализации бэктрекинга пока не решена.

Можем ли мы решить, что `>L{LDROP}_1 == >L{LDROP}_2`? С некоторыми оговорками, да. Если мы постулируем, что `>L{LDROP}_1 == >L{LDROP}_2`, мы примем на себя обязательство не попадать в ситуации, где это не так, то есть рассматривать только фрагменты кода, не зависящие от метода реализации L-стека. *Корректность замены* `>L{LDROP}_1` *на* `>L{LDROP}_2` *будет обеспечиваться за пределами формализма.* В итоге получится гибридная система, состоящая из формальной и неформальной частей. Этого может быть достаточно для наших целей. Если существуют две реализации бэктрекинга, имеет смысл писать программы, работающие на обеих реализациях, то есть программы, не отличающие `>L{LDROP}_1` от `>L{LDROP}_2`.

В многозадачной системе каждая задача должна иметь свой экземпляр LP.

Свойства предложенной реализации L-стека:

1) стек реализован как связный список;
2) память под элементы списка отводится на стеке возвратов;
3) снятие со стека — это исключение из списка и не обязательно освобождение памяти;
4) LP указывает на последнюю запись;
5) записи могут иметь любой размер, не обязательно один и тот же.

3. Локальные переменные

Описанная выше реализация L-стека позволяет использовать указатель L-стека LP еще и как указатель фрейма локальных переменных. Слово CONT, временно снимающее верхнее значение с L-стека, просто временно исключает запись из списка.

Следует отметить следующие свойства приведенной ниже реализации локальных переменных.

1) Во всех трех возможных случаях — при создании фрейма локальных переменных, при исполнении пролога откатного слова, и при исполнении пролога откатного слова с созданием фрейма локальных переменных — на L-стек добавляется ровно одна запись.
2) Слово EXIT не занимается освобождением памяти, отведенной под стек локальных переменных. EXIT просто передает управление коду, адрес которого снимается со стека возвратов. Слова, отводящие память на стеке возвратов

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

под фрейм локальных переменных, оставляют на стеке возвратов адрес кода, отвечающего за освобождение этой памяти.
3) Синтаксис { *имена* } используется для создания фрейма локальных переменных и для описания локальных переменных. В код добавляются слово `_init_loc` и размер фрейма локальных переменных.
4) Синтаксис pro{ *имена* } используется для создания фрейма локальных переменных с исполнением пролога откатного слова и для описания локальных переменных. В код добавляются слово `_init_loc_PRO` и размер фрейма локальных переменных.
5) Пролог откатного слова, не использующего локальные переменные, реализуется словом `PRO`.

```
$0c CONSTANT _loc-offset
: _fetch-local LP @ + @ ;    ( loc-id -- val )
: _store-local LP @ + ! ;    ( val loc-id -- )
: _init_loc ( -- ) ( I: ra0 ra[size] -- frame size ra0 lp-old
[<--lp] ra2 ra+)
  R> R>            ( ra ra0 ) ( R:  )
  OVER @           ( ra ra0 size ) ( R:  )
  RP@ OVER - RP!   ( ra ra0 size ) ( R: frame)
  >R >R            ( ra ) ( R: frame size ra0)
  LP@ rel>R
  RP@ LP!          ( ra)( R: frame size ra0 lp-old [<--lp] )
  CELL+            ( ra+)( R: frame size ra0 lp-old [<--lp] )
   ENTER           ( R: frame size ra0 lp-old [<--lp] )
  relR> LP!        ( ra0 ) ( R: frame size )
  R> R> RP@ + RP!  ( ra0 )
  >R ;
: _init_loc_PRO
  R> R>            ( ra ra0 ) ( R:  )
  OVER @           ( ra ra0 size ) ( R:  )
  RP@ OVER - RP!   ( ra ra0 size ) ( R: frame )
  >R >R            ( ra ) ( R: frame size ra0 )
  LP@ rel>R
   RP@ LP!          ( ra)( R: frame size ra0 lp-old [<--lp] )
  CELL+            ( ra+)( R: frame size ra0 lp-old [<--lp] )
\ ( frame size ra0 lp-old [<--lp] ra2 ra -- frame size ra0 lp-
old [<--lp] ra2' ra )
   ENTER
  relR> LP!        ( ra0 ) ( R: frame size )
  R> R> RP@ + RP!  ( ra0 )
  DROP ;
```

Описание прочих аспектов реализации локальных переменных выходит за рамки данной статьи; мы ограничимся двумя примерами, демонстрирующими соответствие между исходным текстом и порождаемым кодом (слово `see` — декомпилятор).

```
:t { a b } b a ;
: tt pro{ a b c } a CONT b CONT c CONT ;
see t
```


Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

```
: t  _init_loc [14] LIT [c] _store-local LIT [10] _store-local
LIT [c] _fetch-local LIT [10] _fetch-local EXIT  ok[Hex]
see tt
: tt  _init_loc_PRO [18] LIT [c] _store-local LIT [10] _store-
local LIT [14] _store-local LIT [14] _fetch-local CONT LIT [10]
_fetch-local CONT LIT [c] _fetch-local CONT EXIT  ok[Hex]
```

4. Оператор отсечения

Описанный в работе [1] оператор "структурный cut" отличается тем, что:
1) явно отмечаются и начало и конец области, внутрь которой откат не делается;
2) слова, отмечающие начало и конец "отсекаемой" области, должны быть парными;
3) соответствие между ними определяется во время исполнения;
4) оператор cut можно закрыть и без удаления откатной информации.

В [1, 7] оператор отсечения cut реализовывался словами:

```
\ оставить пометку
: CUT:    R> RP@ >L ENTER LDROP ;
\ удалить откатную информацию и пометку
: -CUT    R> L> RP! >R ;
\ убрать только пометку, при откате восстановить
: -NOCUT  R> L> >R ENTER R> >L ;
```

Нетрудно видеть, что здесь пометка оставляется на вершине L-стека. Если мы используем локальные переменные, то верхняя запись L-стека одновременно является фреймом локальных переменных. Если мы поместим отметку на вершину L-стека, то доступа к локальным переменным не будет. Поэтому пометку следует помещать под вершину L-стека. Правило балансировки остается тем же, что и для случая помещения отметки на вершину L-стека: пользоваться L-стеком можно только после того, как пометка снята (словом `-CUT` или `-NOCUT`).

Перепишем слово `CUT`:

```
CUT: == [r R> RP@ >L ENTER LDROP ; '] ==
== [r R> RP@ >L 'r[ LDROP ; ] >R ; '] ==
== [r R> RP@ >L{LDROP} >R ; '] ==
== RP@ >L{LDROP}

-NOCUT == [r R> L> >R ENTER R> >L ; '] ==
== [r R> L> >R 'r[ R> >L ; ] >R ; '] ==
== L> >R 'r[ R> >L ; ] ==
== L>{LREST} DROP

-CUT == [r R> L> RP! >R ; '] == L> RP!
```

В [9] нет правил для `RP!` , однако

```
-CUT == L>{LREST} RP!
```


Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

так как в данном случае, очевидно, что откатная информация, оставленная словом
`L>{LREST}`, будет удалена вместе с остальной откатной информацией.

Так как верхний элемент L-стека трогать нельзя (это еще и указатель блока локальных переменных), будем использовать слова, осуществляющие доступ ко второму элементу стека. Будем считать, что M-стек – это L-стек без верхнего элемента. Итак,

```
CUT: == RP@ >M{MDROP}
-CUT == M>{MREST} RP!
-NOCUT == M>{MREST} DROP
```

Правда, теперь нам необходимо, чтобы перед выполнением слова `CUT`: на L-стеке был хотя бы один элемент.

Свойства оператора `cut`:
1) перед выполнением слова `-CUT` или `-NOCUT` состояние L-стека должно быть тем же, что и после выполнения слова `CUT`:
2) после выполнения слова `-CUT` состояние стека возвратов и L-стека - то же, что и перед выполнением слова `CUT`:
3) после выполнения слова `-NOCUT` состояние L-стека - то же, что и перед выполнением слова `CUT`:

Замечание. В принципе, можно использовать и отдельный стек.

5. Оператор цикла AMONG

Оператор `AMONG` [13] позволяет организовать цикл с откатным телом. Он имеет вид:

`AMONG` *итератор* `EACH` *тело* `ITERATE`

Для каждого значения, вырабатываемого *итератором*, выполняется *тело, итератор* получает управление и вырабатывает новое значение после успеха *тела*. (Неуспех тела приводит не к откату в итератор, а к восстановлению его предыдущего состояния и дальнейшему откату.) Когда *итератор* больше не может выработать нового значения, управление передается за `ITERATE`. Без конструкции `AMONG` мы передавали бы управление итератору не при успехе тела, а при откате из тела.

Методика реализации использует то, что каждый раз при успехе тела на стеке возвратов создается новая копия итератора, в которую и делается откат. При откате же из тела состояние итератора восстанавливается посредством возврата к предыдущей копии.

Замечание. Предполагается, что состояние итератора определяется соответствующим участком стека возвратов. Если это не так, то нельзя считать, что при откате состояние итератора восстанавливается. С другой стороны, взаимодействие тела и итератора через общие структуры данных является признаком того, что конструкция `AMONG` действительно нужна.

Пример 1: слово reverse1 получает адрес (execution token) итератора, оставляющего на стеке одно значение, и выдает эти значения в обратном порядке. Слово `UNAMONG` аналогично слову `UNLOOP` для цикла `DO...LOOP` — оно

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

снимает с L-стека управляющие параметры цикла AMONG, позволяя выполнить слово CONT.

```
: reverse1 PRO AMONG EXECUTE EACH >R BACK R> UNAMONG CONT
TRACKING ITERATE ;
```

Пример 2: слово test распечатывает все подмножества множества {1,2,3}. Слово subs оставляет элементы множества на стеке, они распечатываются словом .S (предполагается, что стек изначально пуст).

```
:elems PRO 4 1 DO I CONT LOOP ;
: subs PRO AMONG elems EACH {| ( оставить) || DROP |} ITERATE
CONT ;
: test subs .S ;
```

Чтобы реализовать конструкцию AMONG, мы переходим к использованию относительных адресов в указателях на элементы стека возвратов. Это избавляет от необходимости корректировки всех указателей при копировании участков стека возвратов, однако указатель в последнем элементе L-стека на перемещенном участке будет содержать неверное значение. Поэтому перед запуском итератора на L-стек помещаются два элемента, а перед передачей управления телу цикла эти элементы снимаются с L-стека. Обойтись одним элементом нельзя, так как слово >M{MDROP} "подсовывает" значение под вершину L-стека. Смещения этих двух элементов в перемещаемом фрагменте стека возвратов известны, так что в последний на перемещенном участке элемент L-стека легко можно записать верное значение.

Отметим, что реализация локальных переменных также должна использовать относительные адреса (т.е. во фрейме локальных переменных должен храниться не адрес предыдущего фрейма, а смещение до него). Оператор cut также должен использовать относительные адреса.

Пользоваться локальными переменными в части между AMONG и EACH нельзя - во-первых, в этой части цикла адрес фрейма не является верхним элементом L-стека, во-вторых, такое ограничение имеет смысл, поскольку эти переменные находятся вне копируемой области и их значения не могут быть восстановлены при откате.

Конструкция AMONG — вещь достаточно экзотическая. Автору довелось применить ее только один раз: в программе, которая проводила синтаксический анализ структуры предложения на русском языке, возникла подзадача перебора основных деревьев графа. (Программа распечатывала древовидные структуры, соответствующие возможным смыслам предложения и требовала описания всех использованных в предложении словоформ; полного словаря словоформ у автора нет.)

```
\ хранение адресов в переместимой форме
: rel>R RP@ - R> SWAP >R >R ;    ( x -- ) ( r: -- x' )
: relR> R> R> SWAP >R RP@ + ;    ( -- x ) ( r: x' -- )
: rel@  DUP @ + ;                ( addr -- x )
: rel!  TUCK - SWAP ! ;          ( x addr -- )

: LMSHUT{LMOPEN}
  R>
```


Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

```
    LP@ rel>R RP@ rel>R RP@ LP!
    BACK RDROP relR> LP! TRACKING
  >R ;
: LMOPEN{LMSHUT}
  R>
    LP@ rel>R
    BACK relR> LP! TRACKING
    LP@ rel@ rel@ LP!
  >R ;
: A>M{MDROP}  ( x --> ) ( l: y --> x y )  (  <--  ) ( l: y <--
x y )
  R> SWAP
    rel>R MP@ rel>R RP@ LP@ rel!
    BACK relR> LP@ rel! RDROP TRACKING
  >R ;
: RPA>M{MDROP}
  R>  RP@ 3 CELLS - A>M{MDROP}  >R ;
: AM>{MREST}  ( --> x ) ( l: x y --> y )  (  <--  ) ( l: x y <-
- y )
  R>
    MP@ rel>R  BACK relR> LP@ rel! TRACKING
    MP@ DUP  rel@ LP@ rel!  CELL+ rel@
  SWAP >R ;
: AM@ MP@ CELL+ rel@ ;          ( -- x )

: PRO R> R> >L{LDROP} >R ;
: CONT L>{LREST} >R ;

: CUT:    R> RP@ A>M{MDROP} >R ;
: -CUT    R> AM>{MREST} RP! >R ;
: -NOCUT  R> AM>{MREST} DROP >R ;

\  AMONG                  ...  EACH  ...  ITERATE
\ порождается код:
\ (among) (among>) {addr} ... (each) ... (iterate) addr:
: (AMONG) R> RPA>M{MDROP} DUP >R TOKEN+ REF+ LMSHUT{LMOPEN} >R
;
: (AMONG>) R> AM>{MREST} DROP REF@ >R ;
: (EACH) R> LMOPEN{LMSHUT} RP@ BACK AM@ RP! TRACKING A>M{MDROP}
>R ;
: (ITERATE) RDROP AM>{MREST} AM>{MREST} RPA>M{MDROP}
  OVER - RP@ OVER - RP! RP@ SWAP MOVE
  LP@ AM@ 2 CELLS - rel! ;
: FINIS     RDROP AM>{MREST} DROP AM@ CELL- @ >R ;
: UNAMONG   R> AM>{MREST} AM>{MREST} 2DROP >R ;
: AMONG    ?COMP COMPILE (AMONG) COMPILE (AMONG>) >MARK 207 ;
IMMEDIATE
: EACH    ?COMP 207 ?PAIRS COMPILE (EACH) 208 ; IMMEDIATE
:  ITERATE  ?COMP  208  ?PAIRS  COMPILE  (ITERATE)  >RESOLVE  ;
IMMEDIATE
```

6. Применимость для алголоподобных языков

Труды СПИИРАН. 2002. Вып. 1. Т. 1. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2002. Issue 1. V. 1. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

Аналогичную методику автор применил в экспериментальной реализации бэктрекинга для языка Си. Полной аналогии, однако, нет: в языке Си выход из процедуры и откат не могут быть одним и тем же действием, так как return, в отличие от `EXIT`, освобождает фрейм локальных переменных.

Наибольшая трудность связана с перемещением адресов при реализации конструкции `AMONG`; пока что эта задача не решена.

В настоящий момент реализацию бэктрекинга для Си нельзя назвать серьезной: это надстройка над одной из версий компилятора GNU C++, ни коим образом не рассчитанного на использование подобных методов.

7. Заключение

Бэктрекинг позволяет ввести в язык еще один вид модульности — модули, отвечающие за перебор.

В данной статье описана еще одна, после [1], методика реализации механизма откатов в Форте. Ее отличительной особенностью является использование только одного стека, что важно для многозадачных систем (так как не в каждой задаче нужен бэктрекинг), и совместимость с механизмом локальных переменных.

Поддержка цикла `AMONG` ведет к дополнительным накладным расходам из-за использования относительных указателей. Если поддержка цикла `AMONG` не нужна, реализация может быть чуть быстрее.

Аналогичную методику автор применил в экспериментальной реализации бэктрекинга для языка Си. Самая сложная из возникающих при этом проблем связана с перемещением адресов в конструкции `AMONG`.

Литература

[1] *Гасаненко М. Л.* Новые синтаксические конструкции и бэктрекинг для языка Форт // Проблемы технологии программирования. — СПб: СПИИРАН, 1991-93, с. 148–162.
`http://www.forth.org.ru/~mlg/BacFORTH-88/BF-diplom.html`

[2] *Баранов С. Н., Ноздрунов Н. Р.* Язык Форт и его реализации. — Л.: Машиностроение, 1988 — 160 с.
`http://www.netlib.boom.ru/books/forth000.html`
`http://www.forth.org.ru/~cactus/library.htm`

[3] *Rodriguez B.* A BNF Parser in Forth. ACM SIGForth Newsletter, vol. 2, no. 2, December 1990, p. 13–18.
`http://www.forth.org/literature/bnfparse.html`

[4] *Rodriguez B.* Rules Evaluation through Program Execution. Proc. of the 1990 Rochester Forth Conf., 1990, p. 123–125.

[5] *Charlton, Gordon.* "FOSM, A FOrth String Matcher". 1991 FORML Conference Proceedings, euroFORML'91 Conference, Oct 11-13, 1991, Marianske Lazne, Czechoslovakia, Forth Interest Group, Inc., Oakland, USA, 1992, p. 313-329.

[6] *Jakeman, Chris.* Portable Back-tracking In ANS Forth. Proc. of the FORML'96 conf.
`ftp://ftp.taygeta.com/pub/Forth/Applications/fosm1v1.zip`

[7] *Gassanenko, M. L.* BacFORTH: An Approach to New Control Structures. Proc. of the Euro-Forth'94 conference, 4-6 November 1994, Royal Hotel, Winchester, UK p. 39–41.
`http://www.forth.org.ru/~mlg/ef94/ef94-2-paper.txt`

[8] American National Standard for Information Systems — Programming Languages — Forth. Document ANSI X3.215-1994, X3J14 Technical Committee, 1994, American National Standards Institute.

[9] *Gassanenko, M. L.* Formalization of Return Addresses Manipulations and Control Transfers. Proc. of the euroFORTH'95 conference, 27-29 October 1995, International Centre for Informatics, Dagstuhl Castle, Germany, 18 p.

http://www.forth.org.ru/~mlg/ef95/ef95-2-paper.txt
http://www.forth.org.ru/~mlg/ef95/ef95-2-talk.txt

[10] *Gassanenko M. L.* 1996. Formalization of Backtracking in Forth // Proc. of the euroFORTH'96 Conf., 4–6 October 1996, Hotel Rus, St.Petersburg, Russia, 26 p.
http://www.forth.org.ru/~mlg/ef96/ef96-1-paper.txt

[11] *Гасаненко М. Л.* Механизмы исполнения кода в открытых расширяемых системах на основе шитого кода. Диссертация на соискание ученой степени кандидата физико-математических наук. — СПб, 1996. — 155 с.

[12] *Gassanenko M. L.* 1999. Threaded Code Execution and Return Address Manipulations from the Lambda Calculus Viewpoint. On-line proc. of the EuroForth'99 conf. at http://dec.bournemouth.ac.uk/forth/euro/ef99.html.
http://www.forth.org.ru/~mlg/ef99/gassanenko99b.pdf

[13] *Гасаненко М. Л.* Расширение возможностей перебора с откатом (бэктрекинга) // Информационные технологии и интеллектуальные методы. — СПб: СПИИРАН, Изд. ТОО "Анатолия", 1997/ — с. 23-35.
http://www.forth.org.ru/~mlg/BacFORTH-90/pgasmil1-96.html