Backtracking and Constraints in Tcl-BC

Dayton Clark and David M. Arnow
Dept. of Computer and Information Science
Brooklyn College

Backtracking and Constraints in Tcl-BC

Dayton Clark and David M. Arnow
Dept. of Computer and Information Science
Brooklyn College
{dayton,arnow}@sci.brooklyn.cuny.edu

ABSTRACT: Tcl is extended to include backtracking controls and a facility for asserting constraints. This increases its flexibility in several domains and opens the possibility for its use in constraint-based logic programming applications and, conversely, applying constraint-based logic programming techniques to more traditional script language domains. The backtracking extensions are written entirely in Tcl itself, a fact which speaks well for Tcl's own extensibility.

1. Scripts and Controls

A good scripting language provides flexible control, data manipulation and convenient access to external facilities. The vast activity that Tcl [Ousterhout, 1990] has spawned speaks to its own merit in this regard [Tcl/Tk Workshops 1993, 1994, 1995]. Much of this activity has focused on extending the language, taking advantage of Tcl's elegant support for that endeavour. These extensions have primarily focused on extending Tcl's access to external resources. The most famous of these have brought Tcl to X [Ousterhout, 1990], to greater process and i/o control [Lehenbauer and Diekhans, 1992; Libes, 1990a, 1990b, 1991, 1993] and to TCP/IP [Smith et al., 1993a], and to audio and multimedia control [Jordan, 1994; Payne, 1993; Smith et al., 1993b; Duval, 1994]. A second group of efforts, most notably [incr Tcl], have sought to provide improved structuring facilities, primarily through object-oriented programming support [Braverman, 1993; Christopher, 1993; Howlett, 1994; McLennan, 1993 and 1994; Menges and Ladd, 1994]. Apart from this group, there have been few modifications of the language itself. To our knowledge, there have been no substantial extensions of Tcl's control structures, although Ousterhout explicitly invites practitioners to do so [Ousterhout, 1993].

In this paper, we present an *intension*[*], that is, an inward extension of Tcl, that derives a new control mechanism, backtracking, from Tcl's unique internal structure. We call this intension Tcl-BC[†] (Tcl with backtracking and constraints). We start by considering at length the motivation for making backtracking available in the language, and then discuss the details of the intension in section 3, providing a few examples in section 4. In section 5, we describe our implementation and then close with a few thoughts about where our work will lead and some questions that it raises.

*. Our neologism, intended to express the idea that we are bringing forth a facility that was always implicit within Tcl. Thus, rather than extending Tcl outward to make external facilities available [as in, for example, Kenny et al., 1993; Richardson, 1993; Smyth, 1994; and Theobald, 1993], we are extending Tcl *inward* to make more available something that was, in a sense, always there.

†. That "BC" are the initials of our employer is purely coincidental.

2. Wanted: Just One More Control Mechanism

Backtracking as a mathematical concept predates electronic computers by at least 60 years [Lucas, 1891] and is, of course, widely used as a programming technique in artificial intelligence and operations research [see, for example, Schalkoff, 1990]. The essential idea is that at some point in the computation a set of choices is encountered, only some of which may turn out to be viable. The choices usually involve the assignment of a value to a variable. The viability of the choices however can only be determined later in the computation. Backtracking allows the choices to be made, in some sequence, and with each choice, the computation proceeds, until "failure" (presumed lack of viability of the choice) is encountered. At that time, control returns to the *choice point*, some, *but not all*[‡], of the program state is restored, and the next choice in the sequence is made. When multiple choice points are involved, failure always returns to the most recent choice point encountered. If and when all the choices of a choice point have been exhausted, another failure is considered to have taken place, relative to the preceding choice point.

Backtracking has some of the flavor of `setjmp`/`longjmp`, but for a number of reasons, not the least of which is the fact that choice points may occur within procedures whose activation has ended, it involves a more complicated state restoration/preservation.

‡. It is because *not* all the program state— including external side effects— is restored that computational progress is made possible.

Figure 1: Backtracking Through Lunch

```
choose (sandwich)
       cheese, pastrami, avocado
choose (sideorder)
       fries, yogurt, beef-barley soup
if (!kosher(sandwich,sideorder))
       fail
print sandwich, sideorder
fail
```

On the left is a sketch of a program that uses backtracking to enumerate all kosher* lunch combinations (within the confines of the rather limited menu!). The first, conditional, failure prevents non-kosher combinations from being printed. The failure at the end forces control back to a choice point in order that *all* combinations be found.

If the kosher predicate is computationally expensive, then in fact this would be a reasonable application of backtracking Tcl. Otherwise it is just a cute, but not very practical, example.

*. For the purpose of this example, we define kosher as simply not mixing meat and dairy products.

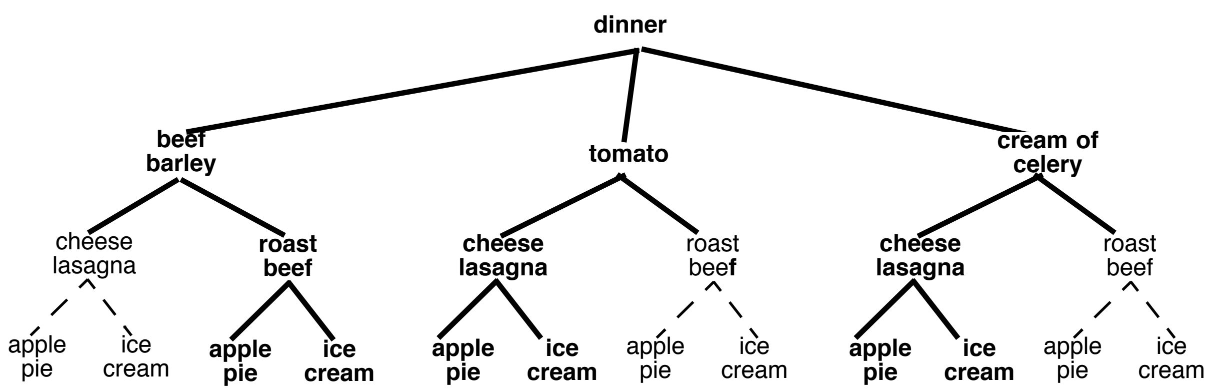
Although it is an elegant means of specifying a progression through a search tree, backtracking in itself does little to facilitate algorithmic efficiency. Search trees involving multiple choices tend to be combinatorially explosive. What is needed is a mechanism for pruning, that is generating failures *before* one reaches the leaves. One such mechanism is *constraints*: conditions that are applied before or during the search which, if violated at any time, cause immediate failure (i.e. return to the most recent choice point for the next choice).

Figure 2: Backtracking Through Dinner— With Constraints

```
constrain (not dairy(entree) and meat(soup))
constrain (not meat(entree) and dairy(soup))
choose (soup)
       beef-barley, tomato, cream of celery
choose (entree)
       cheese lasagna, roast beef
choose (dessert)
       ice cream, apple pie
if (!kosher(soup,entree,dessert))
       fail
print soup, entree, dessert
fail
```

In this example, the program asserts two constraints, which the backtracking mechanism will use to *prune* the search tree induced by the choice points. Only one failing leaf node (beef-barley, roast beef, and ice cream) is actually reached.

The pruned sub-trees are shown below in italicized, non-bold font with dashed edges. Note that these constraints reduce the number of leaves visited by a half.



Despite its utility, backtracking is usually not implemented at the language level, except in the Prolog family of languages. This is because:

- implementing backtracking using conditionals and recursion is a widely understood programming technique;
- backtracking requires an at least implicit distinction between memory objects that are restored to a prior state and those that are not— a distinction that does not come naturally in many languages;
- backtracking involves substantial interaction with the program memory state, complicating the semantics of the language and increasing run-time overhead and compiler complexity.

Backtracking makes great sense in a scripting language like Tcl, however. This is because:

- the purpose of a scripting language is often for fast prototyping or gluing other programs— widely understood or not, script-writers are understandably not eager to re-invent the backtracking wheel;
- scripting languages tend to use implicit declarations— distinguishing between restorable and non-restorable memory objects can be just a matter of listing one set or the other;
- the use of scripting languages for fast prototyping or gluing other programs makes their own speed, and therefore the run-time cost of backtracking, less important than it would be in other contexts.

Finally, there are several characteristics of Tcl that especially invite an implementation of backtracking. First, is the well-known property of scripts as first class objects in Tcl and the availability of **eval**. Second is the curious fact that the Tcl run-time memory organization is not simply a global heap plus a stack of activation records. Consider, for example:

```
#! /usr/local/bin/tclsh

proc Mid {val track} {
   global inMid
   incr inMid
   if {$val > 0} {
      uplevel 1 Mid [incr val -1] [append track ":Mid"]
      Bot [append track ":Mid"]
   }
   incr inMid -1
}

proc Bot {track} {
   global inMid
   puts "Enter Bot --  # in Mid currently = $inMid"
   puts "   up 0 levels: level#=[info level ] ([info level 2])"
   puts "   up 1  level: level#=[uplevel 1 info level ]\
      ([uplevel 1 info level 1])"
   puts "   up 2 levels: level#=[uplevel 2 info level ] (Main)"
}

set inMid 0

Mid 3 "Calling-Trace:Main"
```

The output of this program is shown below. The calling-trace printouts (in the output) correspond to the traditional stack of activation records model and reflect the dynamic history of procedure invocation. The actual environment in which **Bot** executes is given in the printouts of the levels. The next figure illustrates both perspectives when the first entry to **Bot** takes place.

```
Enter Bot --  # in Mid currently = 3
   up 0 levels: level#=2 (Bot Calling-Trace:Main:Mid:Mid:Mid:Mid)
   up 1  level: level#=1 (Mid 1 Calling-Trace:Main:Mid:Mid)
   up 2 levels: level#=0 (Main)
Enter Bot --  # in Mid currently = 2
   up 0 levels: level#=2 (Bot Calling-Trace:Main:Mid:Mid:Mid)
   up 1  level: level#=1 (Mid 2 Calling-Trace:Main:Mid)
   up 2 levels: level#=0 (Main)
Enter Bot --  # in Mid currently = 1
   up 0 levels: level#=2 (Bot Calling-Trace:Main:Mid:Mid)
   up 1  level: level#=1 (Mid 3 Calling-Trace:Main)
   up 2 levels: level#=0 (Main)
```

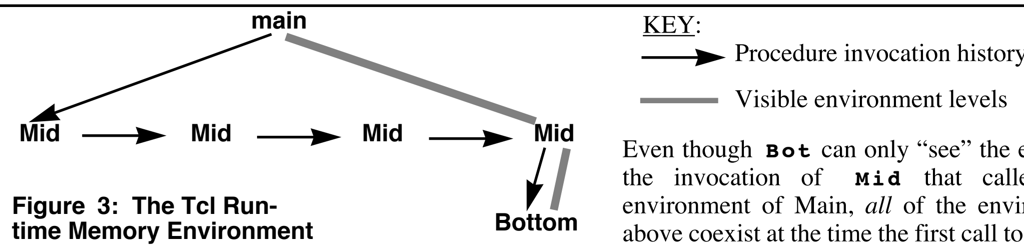


Figure 3: The Tcl Run-time Memory Environment

Even though **Bot** can only "see" the environment of the invocation of **Mid** that called it and the environment of Main, *all* of the environments shown above coexist at the time the first call to **Bot** is made.

Thus there is implicitly a tree of environments in a Tcl computation. We will exploit this in our implementation, as described below in section 5.

2.1 Operations research problems

A substantial source of inspiration to us is the success of the language 2LP [McAloon and Tretkoff., 1995 and 1996]. 2LP combines imperative programming with logic programming facilities (backtracking and constraints) and a linear programming internal object. Compiled and highly efficient, 2LP focuses on a particular domain, and is therefore less general and flexible than C or Tcl. Among other applications, it has been used to implement a scheme for parallel integer goal programming applied to such classical operations research problems as job shop scheduling and the capacitated warehouse problem [Arnow et al. 1995]. The key to 2LP's success lie in these elements:

- an imperative programming model
- a logic programming control— in particular backtracking with constraints
- access to a linear programming solver
- articulation with C

The possibility of utilizing Tcl-BC in conjunction with a standard mathematical modeling package such as AMPL [Fourer et al., 1994], linked to a linear programming solver such as Cplex, is not far from our thoughts. With the addition of access to tools like that, Tcl-BC could function as a highly extensible (as always) state of the art operations research system. Although Tcl's control could not compete in efficiency with 2LP, its advantages would lie in a greater ability to be linked with external resources (data filters, linear programming solvers, distributed programming extensions, etc.) and its greater programming flexibility.

2.2 Agents and backtracking with constraints

Another impetus for this work is the current interest in Tcl as a vehicle for (hopefully) intelligent knowledge agents roaming the internet [Johnson, 1993; Ott, 1994; Ousterhout, 1995]. Much of what internet agents will do is walk through what amount to search trees generated by choice points. Many of the edges in these trees will involve sending a Tcl script to a remote site for execution— a relatively time-consuming operation. Acting intelligently requires the ability to work within constraints, both prior (as in "don't buy anything that has less than a 1 year warrantee") and dynamic (as in "don't investigate buying a system whose first two components are more expensive that an already known complete system"). It also requires the ability to *utilize* such constraints to minimize remote script execution.

Backtracking with constraints is a natural way to define such computations. Furthermore, if one views agents as Tcl scripts, it is likely that agents will be produced dynamically by some sort of GUI application. It is easier to automatically generate constraints and backtracking code to define an intelligent search than to try to build it into a more ad hoc mechanism.

2.3 Backtracking: why not just use Prolog?

Using Prolog entails a commitment to logic programming. The experience of 2LP demonstrates the convenience of providing a logic programming control within the context of an imperative programming model. That backtracking in Prolog turns out to be much faster than in Tcl-BC is no surprise and irrelevant— one doesn't choose a scripting language for the sake of speed.

2.4 Good and bad uses of Tcl-BC

The "hello, world" application of backtracking is the 8-queens problem, and we give, in the appendix, its solution in Tcl-BC. It is, however, precisely *not* the kind of application that Tcl-BC is intended for, except as an example of rapid prototyping. This is because the most of the computational effort lies in the making and unmaking of moves— and not in the computing the consequences of moves.

Good uses of Tcl-BC will be found wherever the cost of computing the consequence of a choice greatly exceeds the cost of making and unmaking choices or wherever significant constraints can be applied to prune the search tree, prior to or during the search. Thus, the use of Tcl-BC in constructing agents, where the consequence of a choice will typically involve the remote execution of a process, seems appropriate.

3. The Tcl-BC Extension

The backtracking and constraint extension to Tcl are implemented entirely in Tcl. It is found in a file called **bc.tcl**. Therefore no special installation procedures are necessary, simply **source** the extensions prior to their use.

The parts of the extension visible to the programmer fall into two categories: the commands which extend Tcl and implement backtracking and constraints, and the commands required because of implementation details which replace standard Tcl commands. The former include **bc_state** which specifies the state variables, **bc_fail** which induces backtracking, **bc_loop** and **bc_choose** which define choice points, and **bc_constrain** which specifies constraints. The latter include **bc_eval**, **bc_for**, **bc_foreach**, and **bc_while** which substitute for the corresponding standard Tcl commands around code that includes choice points.

3.1 The backtracking calls: bc_state, bc_choose, bc_loop, bc_constrain, and bc_fail

The **bc_state** operation adds a list of variables (and/or arrays) to the *backtrackable set*, that is, the set of variables and arrays whose values are to be restored upon backtracking. Such variables and arrays are made global implicitly by this operation.

The **bc_fail** operation causes *failure* to take place, i.e forces a backtrack to the most recent choice point, provided there is one. At that point, the values of the backtrackable set will be restored to that point and the next choice will be taken. If there are no more unexplored choice points, then all **bc_eval**s (see below) fail and generates an exception.

There are two ways of establishing choice points: **bc_choose** and **bc_loop**. The form for **bc_loop** is:

bc_loop {*initialization*} {*test*} {*step*} {
  body
}

When a **bc_loop** is encountered, the *initialization* code is carried out and the *test* is evaluated. If the test fails (now or later), then the **bc_loop** fails, causing failure to take place. If the test succeeds, the *body* is executed once. If, during the execution of the body *or any code thereafter*, a failure takes place, the backtrackable set is restored to the state just prior to execution of the body, the step code is executed (generating the next choice), and the test is made again and the process repeats itself.

The form for **bc_choose** is:

bc_choose {*choice1*} ... {*choiceN*}

When a **bc_choose** is encountered, current choice is set to choice1. Only the current choice code (and not the other choices) is executed. If, during the execution of the current choice or thereafter, a failure takes place, then the backtrackable set is restored to the state just prior to execution of the current choice, and the succeeding choice becomes the current choice, providing there is one. If there is none, then **bc_choose** fails, causing failure.

The **bc_constrain** operation associates a condition with a list of variables or array-elements. After the operation, any change to that variable that causes the condition to be false generates an immediate failure. The form for **bc_constrain** is:

bc_constrain {*variable list*} "*condition*"

Note that if we write

```
bc_constrain {x} "\$x == \$y+1";
   # Backslashes needed to postpone $ evaluation.
```

a change in x's value that causes this condition to be false will generate a failure but a change in y's value that does the same will not! This is somewhat different from the usual logic-programming constraint.

3.2 The wrapper calls: **`bc_eval`**, **`bc_for`**, **`bc_foreach`**, and **`bc_while`**

Logically, the five backtracking operations described above are all that is needed and we truly wish we could end this section of the paper here. However, the backtracking extension requires the ability to gain control of the Tcl script itself in order to return to choice points. As a result, four additional calls are needed. They serve as wrappers of Tcl codes that contain backtracking operations.

First, a definition: *backtracking code* is any Tcl script that contains choice points (i.e. **`bc_choose`** or **`bc_loop`**) or that invokes a dynamic chain that includes activations containing choice points.

The **`bc_eval`** operation enables backtracking. Any backtracking code must be contained in an argument to **`bc_eval`**.

`bc_eval` body

where *body* is a Tcl script.

In addition, loops whose bodies contain backtracking code must replace **`for`**, **`foreach`** and **`while`** with **`bc_for`**, **`bc_foreach`** and **`bc_while`** respectively. Apart from enabling backtracking, the syntax and semantics for these replacements are identical to those for the originals.

An important design consideration was that the extension leave as small a footprint on Tcl as possible. For this reason, the substitutes for the basic control commands is regrettable, but in each case required (Section 5 describes the implementation in some detail). Figure 4 summarizes the use of these wrappers.The substitutions are not syntactic sugar, the construct

```
bc_for {...} {...} {...} {
    code with choice points
}
```

is not a replacement for

```
for {...} {...} {...} {
    bc_eval {
        code with choice points
    }
}
```

The difference occurs if the program bactracks to a choice point within the loop. The **`bc_for`** loop retains control and its iterations will continue. The **`for`** loop does not regain control. After the the **`bc_eval`** is completed execution continues with the command after the **`for`** command.

Figure 4: Backtracking and Non-backtracking code

```
source Bc.tcl

proc x {} {
    code with no choice points          Non-backtracking (needs no wrapper)
    bc_fail
}

proc y {} {
    bc_eval {
        code with choice points          Backtracking (needs wrapper)
        x
    }
}

bc_eval {
    bc_for {...} {...} {...} {          Backtracking (needs wrapper)
        code with choice points
    }
}

code with no choice points  Non-backtracking (needs no wrapper)

bc_eval {
    bc_while {...}{
        code with choice points          Backtracking (needs wrapper)
    }
}
```

3.3 Restriction

Variables used in the code that generates choice points in a **bc_loop** cannot be constrained. The example in the next section show how this restriction can be worked with.

4. Example

To demonstrate the look and feel of Tcl-BC, we consider an example of its use in solving the following famous cryptarithmic puzzle*:

```
  SEND
 +MORE
 MONEY
```

We will develop our program in steps. First, we source the essential **bc.tcl** and a useful script that defines the **wordexpr** procedure that we will use later.

```
#!/usr/local/bin/tclsh
source bc.tcl
source wordarithmetic.tcl
```

The **cryptarithm** procedure will search for a solution; in doing so, it will create choice points, so its procedure body must be a **bc_eval**. To represent the values of the letters appearing in the cryptarithmic puzzle, we will use the variables: **s e n d m o r y**. These will change value as the search proceeds, and will need to be restored upon failure and return to an earlier choice point and so they are arguments to **bc_state**. Three other auxiliary variables will be needed as well in the *backtracking set*: **v vv ilet**. There use will be explained below.

```
proc cryptarithm {} { bc_eval {
bc_state v vv ilet
bc_state s e n d m o r y
```

Next we set up an array letters that maps the integers 0 through 7 to the letters in the puzzle.

```
set nlet 0
foreach let {s e n m o r y d} {
   incr nlet
   set letters($nlet) $let
}
```

As arithmetically sophisticated human beings, we can give, in the form of constraints, the program some intuitions that are obvious to us. For example, we know that m must be 1, that o must be 0 or 1 and is therefore 0 and that s therefore is 8 or 9.

```
bc_constrain {s} "\$s>=8"
bc_constrain {e} "\$e<8"
bc_constrain {e} "\$e>1"
bc_constrain {n} "\$n==\$e+1"
bc_constrain {m} "\$m==1"
bc_constrain {o} "\$o==0"
bc_constrain {r} "\$r+\$n+1>=\$e+10"
bc_constrain {y} "\$y>1"
bc_constrain {d} "\$d>1"
```

For each of the letters in the puzzle (the **bc_for**), we must choose an appropriate value (the **bc_loop**). The former does *not* involve a set of choices (every letter in the puzzle must get a value) and so a **bc_for** suffices, but the latter does and therefore requires a **bc_loop** to generate choice points. The constraint that follows prevents the computation from assigning the same value to more than one letter. The constraint is written in terms of the value, **v**, rather than the particular letter **$letters($ilet)** in order to apply to all future values considered. A separate variable, **vv**, is used for generating the choices because of the rule that constraints cannot be applied to such a variable.

```
bc_for {set ilet 1} {$ilet <= $nlet
     {incr ilet} {
     bc_loop {set vv 0} {$vv <= 9}
        {incr vv} {
            set v $vv
            set $letters($ilet) $v
     }
     bc_constrain {v} "\$v != $v"
}
```

It is only after the **bc_for** that we know that values have been assigned to all the letters in the puzzle. The call to **wordexpr** tests whether we have a successful combination. If not we fail, returning to a choice point generated by **bc_loop**. Otherwise, we print our solution and are done.

```
if {! [wordexpr {
        send + more = money} ] } {
      bc_fail
    }
    puts " $s$e$n$d"
    puts "+$m$o$r$e"
    puts "---------"
    puts "$m$o$n$e$y"
} }
cryptarithm
```

*. The puzzle is solved by finding an assignment of a decimal digit to each of the letters S,E,N,D,M,O,R,Y such that no digit is assigned to more than one letter and such that the above sum, with the letters replaced by the digits, is correct.

The figure on the right shows the output of this program and its time on a Sparc10. When run with just one of the initial constraints,

```
bc_constrain {m} "\$m==1"
```

the program consumes about 50,000 seconds (14 hours) of Sparc10 time. When that constraint is removed... well, it is still running!

Figure 5: `cryptarithm` on a Sparc10

```
iskra> time cryptarithm
 9567
+1085
---------
10652
    124.2 real   58.6 user    0.0 sys
iskra>
```

5. Implementation

There are three main aspects to adding the ability to backtrack with constraints to Tcl (or any language):

- the ability to maintain and restore the value of the state variables to their values at selected points (choice points) in the execution,
- the ability to ensure that constraints are not violated, and
- the ability to backtrack the execution itself to the choice points.

In Tcl-BC, the first two aspects are handled in a straight forward manner using the built-in Tcl facilities. The last aspect requires more code and is conceptually more involved, however, it is still done within Tcl itself.

Maintaining state. First, to maintain the state variables we keep a list of the variable names. Then when a choice point is encountered, we create a script which will restore the variables to their current values. The only non-trivial point is that we must make sure that state variables which don't exist when the choice point is encountered are made to not exist when we backtrack to the point.

Constraints. Tcl's ability to trap whenever a variable is modified via (the standard Tcl command)**`trace`** is the core of the implementation of constraints within Tcl-BC. Modification of any constrained variable invokes a command which finds the appropriate constraints, tests them, and causes a failure if the constraint is violated.

It is worth noting that the list of constraints is itself part of the state and is backtracked.

Backtracking. To backtrack the program flow requires the ability to continue execution at a choice point which may be anywhere within a script. The native Tcl **`eval`** takes an entire script as a single argument and provides no way to begin or continue execution at a point within the script. Hence, **`bc_eval`** was required.

`bc_eval` is meant to be functionally equivalent to **`eval`**. In fact, we imagine that future versions of Tcl-BC may "hijack" **`eval`**, replacing it with **`bc_eval`**.

The first thing **`bc_eval`** does is to turn its argument script(s) into a *strip*. The strip is a Tcl list of the first level commands in the script. Only the first level of commands are broken out, a command (such as **`if`** or **`bc_eval`**) which itself contains scripts becomes a single element in the list (see Figure 6, below).

Figure 6: Figure: Scripts and Strips

Script	Strip
`bc_state v vv ilet`	`{`
`bc_state s e n d m o r y`	`"bc_state v vv ilet"`
`set nlet 0`	`"bc_state ..."`
`foreach let {s e n d m o r y} {`	`"set nlet 0"`
`        incr nlet`	`"foreach let ..."`
`        set letters($nlet) $let`	
`}`	
`bc_constrain {s} "\$s>=8"`	`"bc_constrain ..."`
`...`	`...`
`bc_for {set ilet 1} {$ilet <= $nlet} {incr ilet} {`	`"bc_for ...`
`        bc_loop {set v 0 } ...`	
`        ...`	`...`
`}`	`}`
`if {! [ wordexpr ...} {`	`"if {! [ word ...`
`        ...`	`...`
`}`	
`...`	`}`

The script (from the cryptarithm example) is transformed by **`Bc_eval`** to the list on the right, which is known as a *strip*. **`Bc_eval`** then evaluates each of the individual commands.

Since `bc_eval`'s will most frequently be inside of loops and evaluated many times, a list of encountered scripts and the corresponding strips is maintained. Subsequent invocations of `bc_eval` on a script then bypasses the parsing of the script.

Once the strip is available, `bc_eval` steps through the list evaluating each of the commands, applying `eval`. If one of the commands directly or indirectly invokes `bc_eval` then the current strip and the position within that strip is saved in a list. The new script is then converted to a strip and processing continues. When the end of the new strip is reached, processing of the interrupted strip is resumed.

This gives the flow of control within Tcl-BC an unorthodox flavor of machine language type control (complete with a program counter and branches) combined with implicit subroutine calls. In a sense, there are two simultaneous execution flows, the normal Tcl flow and the Tcl-BC flow. Consequently, naive combinations of `bc_eval` and normal Tcl may lead to unexpected behavior (see section 3.3, Restrictions).

Choice points are not explicitly entered by the programmer, they are implicit in the `bc_loop` and `bc_choose` commands. When a choice point is encountered a the current state (variables specified in `bc_state` statements) is saved in the form of a script that will restore the state. This script along with the current evaluation level (Tcl procedure nesting depth), a pointer to the current strip, and the current position within that strip are pushed onto a stack. Execution then proceeds.

When `bt_fail` is encountered, the choice point stack is popped, the evaluation level, strip, and position are restored, the script which restores the state is evaluated and execution proceeds from the restored position.

Currently, our primary difficulty with respect to compatibility with Tcl in general and other Tcl extensions, is the handling of returned values, from commands or procedures. Solving this problem is at the top of our agenda and it should be solved soon.

Typical presentations of backtracking invoke a tree structure to describe the flow of problem. Tcl-BC does not maintain an explicit tree but the list of partially evaluated strips can be viewed as a tree since each *thread* (as we call them) in the list points to a previous thread in the list. Imbedded in this list for a particular problem is the tree normally used to describe the problem.

6. Practical details: compatibility and availability

Compatibility. Tcl-BC is totally written in standard Tcl and in this regard should not interfere with any other extension. (Tcl-BC does reserve two prefixes, `bc_` and `_bc_`, for naming its procedures and variables). However, `bc_eval`'ed and `eval`'ed scripts cannot be freely intermingled (see section 3.3). Furthermore, `bc_eval` does not correctly implement returned values and procedure returns. These problems are the focus of our current work.

Availability. Backtracking Tcl is distributed as an under-1000 line script, along with documentation and examples. The relevant URL is

http://acc6.its.brooklyn.cuny.edu/~arnow.

Alternatively, send email to the authors.

7. Retrospect and Prospects

We view the development of Tcl-BC as double edged research. We are investigating the integration of backtracking into an imperative programming language and we are investigating the extensibility of scripting languages such as Tcl. We feel comfortable that our combination of backtracking and Tcl, while not seamless, shows that a smooth integration is feasible. As we stated at the outset of this paper, much of the work on extending the language Tcl itself has focused on adapting Tcl to be appropriate for writing very large Tcl programs. In contrast, our effort provides a powerful, expressive tool that makes it easier to use Tcl as a coordinating language.

False starts. As we discussed in section 2, the fact that Tcl can support a tree, rather than merely a stack, of activation records encouraged an implementation of backtracking. However, at the outset of this project we were also intrigued by the notion of using Tcl interpreters to represent active choice points. We were deterred from pursuing this by our inability to find a simple way for an interpreter to inherit state information, including position in a script, from another. Had there been a way to *fork* an interpreter, we could have eliminated the wrapper calls described in section 5.2, at the price of implementing the extension in C.

Regrets. We have already stated our concern over the need for wrappers. But these merely reflect a deeper, regrettable but apparently unavoidable aspect of the implementation-- its assembly code-like interpretation, with a program counter and all.

Why doesn't Tcl have.... Given the maturity of Tcl and our intuition that it was *right for this project*, we worked for the most part on the assumption that whatever we needed *must* be available somewhere within Tcl. For the most part this was true, with the exception of the parser. It seemed surprising that, with Tcl scripts having the status of first class objects, there is no Tcl-provided mechanism for parsing them or at least a procedure to turn a script into a list of commands. The built-in command `info complete` performs the most tedious part of the parsing. We are also wondering whether a broader interpreter-creation/state inheritance mechanism would be possible. Besides facilitating an extension like this it might be very useful in threaded systems.

Future goals. We have several immediate goals. The foremost of these is to hijack Tcl's `eval`. This would eliminate the need for wrappers and, if done properly, reduce the likelihood of collisions with other extensions. Secondly, we would like to explore the performance impact of implementing this extension in C. In addition, we have plans to apply Tcl-BC to the applications cited in section 2.

8. References

Arnow, D.M., McAloon, K. and Tretkoff, C.: Parallel integer goal programming. *Proceedings of the 23rd Annual ACM Computer Science Conference*. Nashville, Tennessee, March 1995.

Braverman, M.S.: CASTE: A class system for Tcl. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

Christopher, W.A.: Writing Object-oriented Tcl-based Systems using Objectify. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

Duval, P. and Liao, T.: Tcl-Me, a Tcl Multimedia Extension. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).

Fourer, R., Gay, D. and B. Kernighan, B.: *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, (1993).

Howlett, G.: Packages: Adding Namespaces to Tcl. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).

Johnson, R.W.: Autonomous Knowledge Agents - How Agents use the Tool Command Language. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

Jordan, E.M.: An Environment for the Development of Interactive Music and Audio Applications. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).

Kenny, K.B., Sarachan, B.D., Sum, R.N., and Uejio, W.H.: Tcl/Tk - An Integration Vehicle for the Microwave/Millimeter-Wave Pilot Sites (MMPS. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

Lehenbauer, K. and Diekhans, M.: Extended Tcl: Extended command set for Tcl, *unpublished manual page*, (January, 1992). This was cited by Lehenbauer in paper at one of the recent TclTk workshops. A more useful reference is:
ftp://ftp.neosoft.com/pub/tcl/tclx-distrib/*

Libes, D.: Curing Those Uncontrollable Fits of Interaction. *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, (June, 1990).

Libes, D.: Using expect to Automate System Administration Tasks", Don Libes, *Proceedings of the 1990 USENIX Large Systems Administration Conference (LISA) IV*, Colorado Springs, CO, (October, 1990).

Libes, D.: Expect: Scripts for Controlling Interactive Programs. *Computing Systems 4*,2 (1991).

Libes, D.: Kibitz - Connecting Multiple Interactive Programs Together. *Software — Practice & Experience* 23, 5 (May, 1993).

Lucas, E.: *Recreations Mathematiques* (1891).

McAloon, K. and Tretkoff, C.: *Optimization and Computational Logic*, Series in Discrete Mathematics and Optimization, J. Wiley and Sons, to appear (1996).

McAloon, K. and Tretkoff, C. 2LP: Linear programming and logic programming, *Proceedings of Principles and Practice of Constraint Programming,* edited by V. Saraswat and P. VanHentenryck, MIT Press, 1995, pp 101-116.

McLennan, M.J.: [incr tcl] - Object-Oriented Programming in TCL. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

McLennan, M.J.: [incr Tk]: Building Extensible Widgets with [incr Tcl]. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).

Menges, J. and Ladd, B.: Tcl/C++ Binding Made Easy. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).

Ott, M.: Jodler -- A Scripting Language for the Infobahn. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).

Ousterhout, J.K.: TCL: An Embeddable Command Language. *Proceedings of the 1990 Winter USENIX Conference*, (January, 1990).

Ousterhout, J.K.: An X11 Toolkit Based on the TCL Language. *Proceedings of the 1991 Winter USENIX Conference*. (January, 1991).

Ousterhout, J.: *Tcl and the Tk Toolkit*., Addison-Wesley (1993).

Ousterhout, J.: Scripts and Agents: The New Software High Ground. Invited talk at the *1995 Winter USENIX Conference*, New Orleans, (January, 1995).

Payne, A.C.: Ak: An Audio Toolkit for Tcl/Tk. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

Richardson, D.: Cooperating Applications through Tcl/Tk and DCE. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif., Berkeley, (June, 1993).

Schalkoff, R.J.: *Artificial Intelligence: An Engineering Approach*. McGraw-Hill (1990).

Smith, B.C., Rowe, L.A., and Yen, S.C.: Tcl Distributed Programming. *Proc of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

Smith, B.C., Rowe, L.A., and Yen, S.C.: A Tcl/Tk Continuous Media Player. *Proc. of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

Smyth, D.E.: Tcl and Concurrent Object-Oriented Flight Software: Tcl on Mars. *Proceedings of the 2nd Tcl/Tk Workshop*, New Orleans, (June, 1994).

Tcl/Tk Workshops: 1993, 1994 and 1995. The 3rd workshop was cosponsored by Usenix which will publish the proceedings. On-line proceedings of the 1st and 2nd workshops can be found at these ftp sites: ftp.aud.alcatel.com/tcl/workshop/ 1993/ tcl93-proceedings.tar.gz and ftp.aud.alcatel.com/ tcl/ workshop/1994/1994_workshop.tar.gz

Theobald, D.: Interfacing an Object-Oriented Database System from Tcl. *Proceedings of the 1st Tcl/Tk Workshop*, Univ. of Calif. at Berkeley, (June, 1993).

9. Appendix: 8-queens

The 8-queens problem is the standard illustration of backtracking in AI textbooks. The idea is to place 8 queens on the chessboard so that no queen is attacking any other queen. That amounts to choosing a set of 8 squares from among the 64 squares in the 8x8 chessboard such that no pair of squares lies on the same row, column or diagonal. The latter requirements are the *constraints* of the problem.

```
#!/usr/local/bin/tclsh
source bc.tcl
bc_eval {
   bc_state queen r c row column queen_list
   set queen_list {}
   set row 0
   set column 0
   set row_max 8
   set column_max 8
   bc_for {set queen 1} {$queen <= 8} {incr queen} {
      bc_loop {set r 1} {$r <= $row_max} {incr r} {
           set row $r
      }
      bc_loop {set c 1} {$c <= $column_max} {incr c} {
           set column $c
      }
      lappend queen_list [list $row $column]
      bc_constrain {row} "\$row != $row"
      bc_constrain {column} "\$column != $column"
      bc_constrai n {column} "abs(\$column - $column) != abs(\$row - $row)"
   }
   puts stderr "Results--"
   foreach position $queen_list {
      puts stderr "   row [lindex $position 0] column [lindex $position 1]"
   }
}
```