Dynamic Environments — A Generalization of Icon String Scanning*

Kenneth Walker

TR 86-7

March 3, 1986

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8401831.

Dynamic Environments — A Generalization of Icon String Scanning

1. Introduction

As described in [1], the string scanning operator of Icon [2], ?, sets up a *scanning environment* and evaluates a *matching expression* in that environment. In order to better understand the concept of a scanning environment and to explore variations on string scanning, an extension to Icon has been developed for creating user defined environments.

This research is a step in the development of a pattern-matching laboratory. One goal for the development of this laboratory is a specification system for automatically generating pattern-matching facilities. User defined environments will serve as basis for this specification system.

2. Generalization of Environments

The first step in generalizing a concept is to identify the fundamental attributes of that concept. For a scanning environment these attributes are:

- A scanning environment consists of the set of variables &subject and &pos.
- It is created by a scanning expression, *expr*$_1$? *expr*$_2$.
- The environment is initialized based on the value of *expr*$_1$.
- The new values of &subject and &pos are visible in *expr*$_2$ and are visible to any procedures called in *expr*$_2$.
- A scanning environment can be nested within another scanning environment. The outer values of &subject and &pos are hidden while the nested ones are in existence.

Icon string scanning is designed for problems requiring string analysis and synthesis. Because string analysis usually starts at the left end of the string, the variable &pos is automatically initialized to 1. Icon's string scanning facility includes a set of built-in functions for examining and manipulating a scanning environment. The environment variables are seldom accessed explicitly except by procedures meant to extend this built-in repertoire.

Having a user-defined environment feature requires that users be able to specify a set of their own variables as an environment. An environment declaration is modeled after Icon's record declaration, which also specifies a template for a set of variables. The result of an environment declaration is in an environment constructor, analogous to a record constructor. The environment declaration has the form:

```
envir name (id1, id2, ... , idn)
   build  locals1
          expr1

   setup  locals2
          expr2

   eval   locals3
          expr3
end
```

envir, build, setup, and eval are reserved words. *name* is the name of the environment type and the name of the environment constructor. *id*$_1$, *id*$_2$, ... , *id*$_n$ are the names of the environment variables. The build, setup, and eval clauses associate executable code (*expr*$_1$, *expr*$_2$, and *expr*$_3$) with the environment. Each *locals*$_i$ is an optional set of declarations (local, static, or dynamic) whose variables have a scope local to the corresponding expression, *expr*$_i$. Each of these three clauses is optional. Their use is described below.

The semantics of the ? operator have been changed for use with user-defined environments. The revised ? operator is referred to as the environment operator and an expression of the form, $expr_1$? $expr_2$, as an environment expression. The environment operator is responsible for the scope of environments. When an environment variable is referenced in an environment expression, it must be prepended with an &. For example if nsubj is declared as an environment variable it would be referenced as &nsubj.

The environment operator can emulate Icon's string scanning operator and create a scanning environment, but it also accepts as its first operand an environment created by an environment constructor. The following is an example.

```
envir nscan(subj, posn)
end

procedure ntab(i)
   suspend &subj[.&posn : &posn <- i]
end

procedure main()
   nscan("abcd", 1) ? write(ntab(3))
end
```

An environment, nscan, is defined to have two variables, subj and posn. The procedure ntab operates on instances of this environment. It expects &subj to contain a string and &posn to contain an integer index into the string. It changes &posn to the value of its argument and returns the substring of &subj between the old and new values of &posn. If it is resumed, it restores &posn to its old value and fails. The expression nscan("abcd", 1) creates a nscan environment with &subj equal to "abcd" and &posn equal to 1. The environment operator makes the two environment variables directly visible and executes write(ntab(3)) in this scope. ntab(3) changes &pos to 3 and returns "ab", which is then printed by the write function.

It might be convenient if &posn defaulted to 1. However, in the expression nscan("abcd") it defaults to the null value. The expression in the build clause of the environment is executed as part of the environment constructor. This allows programmers to specify default values or otherwise manipulate the environment variables at the time the enironment is created. The environment variables (with the preceding &) can be accessed in the build expression; however, these are parameters to the environment constructor and have local scope. Any procedures called by the build clause see the environment in effect when the constructor is invoked. &posn can be given a default value as follows:

```
envir nscan(subj, posn)
   build /&posn := 1
end
```

The environment expression from the previous example can now be written nscan("abcd") ? write(ntab(3)).

With the appropriate set of procedures, nscan could be used for Icon string scanning. However, there is already a built-in environment, scan, for this purpose. It has the following definition:

```
envir scan(subject, pos)
   build /&pos := 1
end
```

If $expr_1$ of an environment expression evaluates to a string (or a value that can be converted to a string) rather than an environment, a scanning environment is created implicitly. This means that the expression

   "abcd" ? $expr_2$

is equivalent to

   scan("abcd") ? $expr_2$

An environment is designed to handle a specific class of problems. For this reason, it may be useful for the environment operator to do more than just establish a scope for the environment variables. There may be computations that should always be done at the beginning of $expr_2$ or at the end, or it may be useful if the value of the environment expression is based on the final values of the environment variables rather than the value of *expr₂*. The setup clause and the eval clause are used for these purposes. The setup clause is executed before *expr₂* is evaluated and the eval clause is executed after it is evaluated. The value of the environment expression is determined by the keyword &value. It is set to the value of *expr₂* but may be examined and modified by the eval expression. These features have the effect of replacing *expr₂* by the mutual evaluation:

(*setup*, &value := *expr₂*, *eval*, &value)

where *setup* and *eval* are the expressions from the environment declaration for *expr1*. Care should be taken to insure that *setup* and *eval* do not fail, unless backtracking is intended.

The following example shows a version of string scanning that includes an unanchored mode and string synthesis.

```
envir xscan(subj, anchor, object, posn)
   build {
      /&object := ""
      /&posn := 1
      }
   setup local cur_anch
      if &anchor ~=== 1 then
         (cur_anch := &posn) | (&posn :=  |(*&subj + 1 >= (cur_anch +:= 1)))
      else
         &null
   eval &value := &object
end

procedure ntab(i)
   suspend &subj[.&posn : &posn <- i]
end

procedure append(s)
   suspend (&object <- &object || s)
end

procedure main()
   every write(xscan("abc") ? append(ntab(0)))
end
```

&anchor is a flag. Unless it is set to 1, scanning is done in unanchored mode. The setup and eval clauses can be generators. In this case, the expression

```
(cur_anch := &posn) | (&posn :=  |(*&subj + 1 >= (cur_anch +:= 1)))
```

from the setup clause is resumed when *expr₂* fails. The anchor position, cur_anch, starts at the value of &posn when the environment is put into effect. When the expression is resumed, the anchor position is incremented, &posn is set to it, and *expr₂* is re-evaluated. This process stops when the anchor moves beyond the end of the string.

&object is a synthesized string. It defaults to the empty string when an environment is created. The procedure append modifies &object by appending a string to it. The result of an environment expression using xscan is the value of &object. The output of this program is the four strings:

```
"abc"
"bc"
"c"
""
```

3. Environments as First-Class Objects

Environments can be assigned to variables and manipulated outside of scanning expressions. The environment variables can be referenced using the field reference operator, "." (in this context a variable name is not preceded by an &). In fact, environments can be used like records. An example is:

```
envir like_rec(a, b)
end

procedure main()
   local x
   x := like_rec(1, 2)
   write(x.a)
   x.b := 3
end
```

The real advantage of having environments as first-class objects is that a partially processed environment can be saved so that more processing can be done later. This usage is illustrated in the following examples using string scanning.

At first glance, the string scanning of standard Icon may seem ideal for lexical analysis, but this is not so. Lexical analyzers are usually written to analyze input a token at a time and pass that token back to a calling routine. When input is read a line at a time, this means keeping track of a position in the middle of a line until the next token is requested. When a string scanning expression is exited, the value of &pos is lost, making it awkward to keep track of where the scanning left off. That problem is easily solved with the environment feature; the value of &pos can be retained by retaining the entire scan environment across calls to the lexical analyzer. The following is a simple lexical analyzer.

```
procedure getword()
   local nextword
   static line, WhiteSpace, WordChars

   initial {
      line := scan("")
      WhiteSpace := ' \t\n'
      WordChars := &ucase ++ &lcase ++ '0123456789'
      }

   line ?
      while /nextword do {
         if pos(0) then (nextline() | break)
         tab(many(WhiteSpace))
         if ="#" then
            tab(0)
         else
            nextword := (tab(many(WordChars)) | move(1))
      }
   return \nextword
end

procedure nextline()
   return &subject := read()
end
```

```
procedure main()
   while write(getword())
end
```

line is a static variable whose value is the scan environment containing the current input line. It is used to retain the partially completed string scan across procedure calls. The while loop in the environment expression searches for the next token. nextword is set to that token, when it is found, causing the loop to terminate and the token to be returned from the procedure. When the end of a line is reached, nextline is called. It sets &subject to the next line (implicitly resetting &pos to 1) or fails if there is no more input. When nextline fails, the loop is forced to terminate with nextword equal to the null value so that return \nextword causes getword to fail. This model is easily extended to more realistic lexical analyzers.

Programmers often want to write procedures that take a string and *generate* pieces of that string. The straightforward approach is is to suspend the procedure in the middle of a scanning expression. However, in the current implementations of Icon suspend does not restore the outer scanning environment, so problems arise if the procedure is called in the middle of another scan. The following is an example of this approach:

```
procedure sep(s)
   s ? while not pos(0) do {
      tab(many(' '))
      suspend tab(many(~' ')) \ 1
      }
end
```

To insure that scanning environments are properly maintained the suspend must be moved outside of any scanning expression. This can be done by using an explicit scanning environment and doing the scanning in pieces.

```
procedure sep(s)
   local sc

   sc := scan(s)
   while sc ? not pos(0) do {
      sc ? tab(many(' '))
      suspend sc ? tab(many(~' ')) \ 1
      }
end
```

4. Uses Beyond String Scanning

4.1 Tree Manipulation

The environment feature was created as a generalization of Icon's string scanning. However, it may be used in ways that have nothing to do with string processing. Consider the problem of manipulating binary trees. (This example is designed to demonstrate the use of environments and is not intended to be a realistic tool for tree manipulation.) The following record is used represent nodes in the trees:

```
record node(value, lchild, rchild, parent)
```

A tree environment consists of references to a root node and to the node that is the current focus of attention. The focus of attention defaults to the root when the environment is created. The result of an environment expression using a tree environment is a reference to the root of the tree.

```
envir tree(root, curnode)
   build /&curnode := &root
   eval &value := &root
end
```

By itself this environment is not very useful; it needs a set of procedures to manipulate it. The set of procedures in this example follows a specific protocol. Each procedure returns the value of &curnode unless its function specifically requires some other value. Each procedure preforms data backtracking; that is, if it is resumed any side effects it caused are reversed. &curnode is either a node or the null value. Each procedure that references its fields fails if it is not a node.

Some procedures take a subtree as an argument. This subtree is a node (or the null value) and is assumed to have no parent. The procedure new_root replaces the existing tree and sets &curnode to the new root.

```
procedure new_root(subtree)
   suspend .(&root <- subtree, &root.parent <- &null, &curnode <- &root)
end
```

The procedure at_leaf succeeds if &curnode is at the bottom of the tree.

```
procedure at_leaf()
   if (type(&curnode) == "node" & /&curnode.lchild & /&curnode.rchild) then
      return .&curnode
   else
      fail
end
```

The procedures lchild, rchild, and parent are used for moving &curnode through the tree. They fail if the move cannot be made.

```
procedure lchild()
   if type(&curnode) ~== "node" then fail
   suspend .(&curnode <- \&curnode.lchild)
end

procedure rchild()
   if type(&curnode) ~== "node" then fail
   suspend .(&curnode <- \&curnode.rchild)
end

procedure parent()
   if type(&curnode) ~== "node" then fail
   suspend .(&curnode <- \&curnode.parent)
end
```

The procedure value returns the variable for the value field of the current node.

```
procedure value()
   if type(&curnode) ~== "node" then fail
   return &curnode.value
end
```

The procedures l_add and r_add add children to the current node, replacing whatever was there.

```
procedure l_add(subtree)
   if type(&curnode) ~== "node" then fail
   if \subtree then
      suspend .(subtree.parent <- &curnode, &curnode.lchild <- subtree, &curnode)
   else
      suspend .(&curnode.lchild <- &null, &curnode)
   fail
end
```

```
procedure r_add(subtree)
   if type(&curnode) ~== "node" then fail
   if \subtree then
      suspend .(subtree.parent <- &curnode, &curnode.rchild <- subtree, &curnode)
   else
      suspend .(&curnode.rchild <- &null, &curnode)
   fail
end
```

The procedure del deletes the current node, making its parent the new current node.

```
procedure del()
   if type(&curnode) ~== "node" then fail
   if /&curnode.parent then
      suspend .(&root <- &null, &curnode <- &null)
   else if &curnode.parent.lchild === &curnode then
      suspend .1(&curnode <- &curnode.parent, &curnode.lchild <- &null)
   else
      suspend .1(&curnode <- &curnode.parent, &curnode.rchild <- &null)
end
```

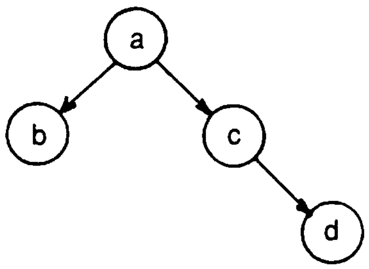
The procedures above are the primative operations on the environment. From them it is possible to build other procedures. As discussed in [2], it is sometimes useful to represent trees as strings. For this example the following recursive notation is used to represent a tree:

label(left_subtree, right_subtree)

label is a string which is the value of the node of the tree (in general the value may be anything, but in this example labels are limited to strings.) *left_subtree* and *right_subtree* are strings of the same format as the tree — the empty string represents the null tree. A tree with one node labeled "xyz" would be represented by:

"xyz(,)"

A tree like:



would be represented by:

"a(b(,),c(,d(,)))"

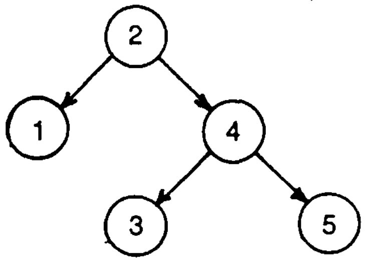
The procedure construct takes a string and returns the root node of the tree (not a tree environment) that the string represents. Note that it uses tree manipulation and string scanning simultaneously.

```
procedure construct(s)
   if s == "" then return &null
   return (
      tree() ? (
         s ? {
            new_root(node(tab(upto('('))))
            move(1)
            l_add(construct(tab(bal(','))))
            move(1)
            r_add(construct(tab(bal(')'))))
            }
         )
      )
end
```

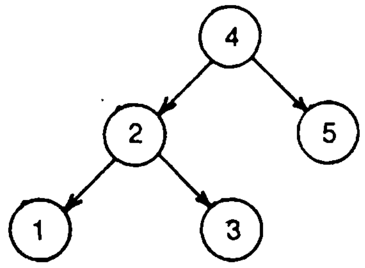
The procedure print is executed in a tree environment and does a preorder print of the subtree rooted at the current node. It prints the value of each node, one value per line, and indents one space for each level of the subtree. print must insure that the current node is the same when it return as when it is invoked, therefore each successful call to lchild and rchild is followed by a call to parent.

```
procedure print(indent)
   /indent := ""
   write(indent, value())
   if at_leaf() then return
   indent ||:= " "
   if lchild() then {
      print(indent);
      parent()
      }
   else
      write(indent, ">>null<<")
   if rchild() then {
      print(indent);
      parent()
      }
   else
      write(indent, ">>null<<")
end
```

The procedure rotate is designed to be used with a sorted tree. It does a left rotation around the current node (for simplicity it assumes that the current node is the root) leaving the tree in order. For example, if it starts with the tree:



the result is:



```
procedure rotate()
   local top, left, left_right

   top := rchild()
   left_right := lchild()
   del()
   left := del()
   r_add(left_right)
   new_root(top)
   l_add(left)
end
```

The right child of the initial root is the root of the new tree. Its left child is the right child of the left child of the root in the new tree. Two dels cut the tree apart leaving the current node back at the root which is the left child of the new root (note that del returns the parent of the node deleted). The entire left subtree is constructed, the new root installed, and the left subtree added on.

The following main procedure shows construct, print, and rotate being used.

```
procedure main()
   tree(construct("2(1(,),4(3(,),5(,)))")) ? {
      write("initial tree:")
      print()
      rotate()
      write("\nrotated tree:")
      print()
      }
end
```

4.2 Context Switching

The tree environment has the same flavor as the scan environment. It consists of a data object and a pointer into the object along with a set of procedures for manipulating the environment. However, environments may be very different. The following example implements context switching between command interpreters.

These interpreters receive commands from standard input and write responses to standard output. One interpreter implements a text buffer. Most input is appended to the buffer, but an * command prints the buffer. In the following sample session, input is printed in italics. The default name for this interpreter is buffer which it prints when it is invoked.

```
***buffer***
this is some input to
the buffer.
*
this is some input to
the buffer.
And this is some more.
*
this is some input to
the buffer.
And this is some more.
```

The second interpreter sends its commands to the operating system (in this case Unix). When it receives a null command it repeats the previous command. The default name for this interpreter is system. The following is a sample session:

```
***system***
pwd
/usr2/kwalker/envir
cat test
This is the contents of the file test.

This is the contents of the file test.
```

Both interpreters recognize ''context'' commands. These begin with "." and are used to create instances of interpreters and to switch between instances. ".b" creates a buffer interpreter and ".s" creates a system interpreter. Either of these commands may be followed by a name for the interpreter. A plain "." will print a list of the existing interpreters. "." followed by a number will switch to one of these existing interpreters. The program starts with a single buffer interpreter.

```
***buffer***
.s
***system***
.banother buffer
***another buffer***
.snamed "to system" interpreter
***named "to system" interpreter***
.
1: ***buffer***
2: ***system***
3: ***another buffer***
4: ***named "to system" interpreter***
.2
***system***
```

The context of an interpreter routine consists of an explicit set of state variables and an execution point. The interpreters in this example execute the same code for each command they interpret. Because a context switch only occurs between commands (it is the result of a command), the execution point at the switch is always the same and does not need to be explicitly retained. This reduces a context to the set of state variables for the interpreter.

Each interpreter is implemented as an environment. The environment variables are the interpreter's state variables. The design of these environments follows a specific pattern. The first environment variable is an identifier; each instance of the environment (and thus the interpreter) may have a different name. The setup clause of the environment prints the identifier.

The eval clause is the body of the interpreter. *$expr_2$* of an environment expression in which these environments are used generates strings which the eval clause interprets; for example, |read() generates strings from standard input. These strings are found in &value. In order to obtain the next string, the eval clause fails. This can be seen by looking at the mutual evaluation used in Section 2 to explain how the setup and eval clauses interact with *expr*$_2$.

(*setup*, &value := *expr*$_2$, *eval*, &value)

Some commands indicate a context switch. For these commands the eval clause succeeds after setting &value to the new context; the surrounding expression must do the actual switch.

The environment buffer implements a text buffer. Any command that starts with a "." is assumed to be a context command and is passed to the procedure context which is shared among all interpreters. If context returns a value, the environment expression will succeed with this value. If the command to buffer is "*" the text buffer is printed. Any other input is appended to the text buffer.

```
envir buffer(b_id, text)
   build {
      if &b_id == "" then &b_id := "buffer"
      &text := ""
      }

   setup write("***", &b_id, "***")

   eval {
      &value ?
         if ="." then
            &value := context()
         else {
            if ="*" then
               writes(&text)
            else
               &text ||:= tab(0) || "\n"
            &fail                          # backtrack for another command
            }
      }
end
```

The environment to_system passes commands to the system. As with buffer, commands starting "." are assumed to be context commands. to_system saves the previous command it sent to the system. If it receives an empty string as a command it resends this previous command.

```
envir to_system(s_id, last_cmd)
   build {
      if &s_id == "" then &s_id := "system"
      &last_cmd := ""
      }

   setup write("***", &s_id, "***")
```

```
      eval {
         &value ?
            if ="." then
               &value := context()
            else {
               if pos(0) then
                  system(&last_cmd)
               else
                  system(&last_cmd := tab(0))
               &fail                          # backtrack for another command
               }
         }
   end
```

The procedure context maintains a list of interpreter environments. It preforms three functions: it prints the list, returns one of the environments or adds an environment to the list. context is called in a scanning environment with &pos positioned at its command string. When it receives an empty command, it goes through the list, prints each index and invokes the corresponding environment with an empty command sequence. The environment identifies itself because the setup clause is invoked, but the environment expression fails before the eval clause is invoked. Note that the environment that calls context is invoked recursively.

If the command to context is a number, it is taken to be an index into the list of environments and context returns the corresponding environment. If the command is a "b" or an "s", a new buffer or to_system environment respectively is added to the list and returned. The remainder of the command line is passed as the first argument to the environment constructor, which is by convention the identifier of the environment.

```
   procedure context()
      local i
      static interpreters     # list of existing interpreter environments

      initial interpreters := []

      if pos(0) then {
         every i := 1 to *interpreters do {
            writes(i, ": ")
            interpreters[i] ? &fail    # get interpreter to identify itself
            }
         fail
         }
      else if i := integer(tab(0)) then
         return interpreters[i] | (write("out of range"), &fail)
      else if ="b" then {
         put(interpreters, buffer(tab(0)))
         return interpreters[-1]
         }
      else if ="s" then {
         put(interpreters, to_system(tab(0)))
         return interpreters[-1]
         }
      else {
         write("invalid command")
         fail
         }
   end
```

The main procedure maintains the currently active environment and ''feeds'' it lines from standard input. The initial environment is created by giving context a "b" command. The environment identifier in the command is the empty string, but the buffer environment will change it to the default of "buffer". The augmented environment operator ?:= is used to change the active environment whenever the environment expression succeeds.

```
procedure main()
   local active

   "b" ? (active := context())
   while active ?:= |read()
end
```

5. Implementation

Environments are implemented by translating environment declarations and environment expressions into standard Icon. The preprocessor to do this translation was constructed using the variant translator system described in [3]. The examples in this section are from the translation of the xscan environment from Section 2.

An environment is represented by a set of global variables, a record, and four procedures. An environment declaration of the form:

```
envir name ($id_1$, $id_2$, ... , $id_n$)
   ...  end
```

is translated into:

```
global K_$id_1$_, K_$id_2$_, ... , K_$id_n$_, Cur_name_

record Envir_name_($id_1$, $id_2$, ... ,$id_n$, Switch_, Setup_, Eval_)

procedure Switch_name_(env)
   ...  end

procedure Setup_name_()
   ...  end

procedure Eval_name_(K_value_)
   ...  end

procedure name(K_$id_1$_, K_$id_2$_, ... , K_$id_n$_)
   ...  end
```

An instance of an environment is an instance of the corresponding record. There is one global variable for each environment variable; these contain the currently visible environment and create the effect of dynamic scope. The names of these global variables are created by prepending K_ and appending _ to the names of the environment variables. There is also a global variable to hold the record for the currently visible environment. The following is the global declaration from the translation of the example.

```
global K_subj_, K_anchor_, K_object_, K_posn_, Cur_xscan_
```

The preprocessor assumes that any non-standard keywords appearing expressions are environment variables and does the appropriate translation. This can be seen in the translation of the procedures ntab and append:

```
procedure ntab(i);
   suspend K_subj_[.K_posn_:K_posn_ <- i];
end

procedure append(s);
   suspend (K_object_ <- K_object_ || s);
end
```

The *constructor* procedure constructs an instance of the environment record. It contains the code from the build clause. The record has a field for each environment variable and one for each of the remaining three procedures.

The constructor procedure uses the values of its parameters to initialize the "variable" fields of the record and uses the switch, setup, and eval procedures of the environment for the values of the Switch_, Setup_, and Eval_ fields.

```
record Envir_xscan_(subj, anchor, object, posn, Switch_, Setup_, Eval_)

procedure xscan(K_subj_ , K_anchor_ , K_object_ , K_posn_)
   {
    /K_object_ := "";
    /K_posn_ := 1
   }
   return Envir_xscan_(K_subj_, K_anchor_, K_object_, K_posn_, Switch_xscan_,
      Setup_xscan_, Eval_xscan_)
end
```

The switch procedure takes as its parameter a corresponding environment record and makes it the currently visible environment. The fields of the record for the previously visible environment are updated to reflect any changes the its environment variables and a reference to this previous record is returned. The switch procedure reverses the switch during backtracking, updating the new environment record to reflect any side affects to the environment variables that were not undone by the backtracking.

```
procedure Switch_xscan_(env)
   local PrevEnv

   # save the previous environment
   PrevEnv := Cur_xscan_
   PrevEnv.subj := K_subj_
   PrevEnv.anchor := K_anchor_
   PrevEnv.object := K_object_
   PrevEnv.posn := K_posn_

   #establish the new environment
   K_subj_ := env.subj
   K_anchor_ := env.anchor
   K_object_ := env.object
   K_posn_ := env.posn
   Cur_xscan_ := env

   suspend PrevEnv

   # save changes to the new environment
   env.subj := K_subj_
   env.anchor := K_anchor_
   env.object := K_object_
   env.posn := K_posn_

   # re-establish the previous environment
   K_subj_ := PrevEnv.subj
   K_anchor_ := PrevEnv.anchor
   K_object_ := PrevEnv.object
   K_posn_ := PrevEnv.posn
   Cur_xscan_ := PrevEnv

   fail
end
```

The setup and eval procedures contain the code from the corresponding clauses. A suspend is used so the code will be resumed during backtracking. In the eval procedure the keyword &value is implemented as the parameter K_value_. The expression from the eval clause is executed as part of a mutual evaluation so the result of the procedure is "&value".

```
procedure Setup_xscan_()
   local cur_anch;
   suspend if K_anchor_ ~=== 1 then
      (cur_anch := K_posn_) | (K_posn_ := |(*K_subj_ + 1 >= (cur_anch +:= 1)))
      else &null
end

procedure Eval_xscan_(K_value_)
   suspend (K_value_ := K_object_, K_value_)
end
```

The model used to implement environment expressions is based on the one used in [1] for string scanning. The following translation is done to environment expressions:

$\tau(expr_1 \text{ ? } expr_2)$ = Remove_(Setup_($\tau(expr_1)$), $\tau(expr_2)$)

where $\tau(expr)$ denotes the translation preformed by the preprocessor. The Setup_ and Remove_ procedures are linked with the translated program along with the code to implement the built-in environment, scan.

```
procedure Setup_(env)

   # if argument can be converted to a string then default to string scanning
   env := scan(string(env), 1)

   suspend 1(env.Switch_(env), env.Setup_())
   fail
end

procedure Remove_(env, value)
   suspend 1(env.Eval_(value), env.Switch_(env))
   fail
end
```

These procedures are responsible for the proper nesting of environments and the environment specific semantics of the environment operator. Most of the actual work is done by the procedures obtained from the environment record. These are stored under the same field names in all environments: Switch_, Setup_ and Eval_, so the Setup_ and Remove_ procedures do not have to know what environment they are working with.

The first step in the evaluation of the translated expression

Remove_(Setup($expr_1$),$expr_2$)

is the evaluation of $expr_1$. It must result in an environment instance or a value that Setup_ can convert to an instance of a scanning environment. Setup_ switches to this environment instance with the call env.Switch_(env) then invokes the setup code specific to that environment with the call env.Setup(). Finally it suspends, passing the outer environment instance, returned by env.Setup(env), to Remove_. Before Remove_ is invoked, $expr_2$ is evaluated and its result is also passed to Remove_. Remove_ invokes the eval code specific to this environment, making $expr_2$'s value available to the eval code. This is done with the call env.Eval_(value). It then switches back to the outer environment instance with the call env.Switch_(env). It suspends making the result of the eval code the result of the environment expression. Because both procedures use mutual evaluation and suspension, these processes can be reversed though backtracking.

The last thing the translator does is generate code to initialize each of the current environment records. The remaining code from the translation of the xscan example is:

```
procedure main();
   Init_()
   every write(Remove_(Setup_(xscan("abc")), append(ntab(0))));
end

procedure Init_()
   Cur_scan_ := Envir_scan_(&subject, &pos, Switch_scan_, Setup_scan_,
      Eval_scan_)
   Cur_xscan_ := Envir_xscan_(K_subj_, K_anchor_, K_object_, K_posn_,
      Switch_xscan_, Setup_xscan_, Eval_xscan_)
end
```

6. Conclusions

In [1] scanning environments are presented as an abstraction to describe the semantics of the string scanning operator, ?. The envir feature described in this report extends this idea in several ways. First, it allows the declaration of environments containing an arbitrary set of state variables. This is useful in the development of more elaborate models for string processing and in the development of models for processing data other than strings.

Second, the envir feature allows environments to be created independent of scanning expressions and assigned to variables. This takes environments out of the realm of the abstract and makes them a concrete part of the language. The ability to retain an environment between executions of environment expressions simplifies some problems of string scanning and can be applied to problems outside of string scanning.

Finally, envir declarations allow procedural code to be associated with each environment type. The build clause is incorporated into the environment constructor and the setup and eval clauses extend the semantics of the environment operator. These clauses enhance the usefulness of environments in exploring extensions to string scanning. As can be seen in the context switching example in Section 4, these clauses can also be used in applying environments to problems far removed from string scanning.

Acknowledgements

Ralph Griswold served as advisor for this project. Both he and Dave Hanson provided numerous helpful suggestions on the presentation of this report.

References

1. K. Walker and R. E. Griswold, *A Pattern-Matching Laboratory; Part I — An Animated Display of String Pattern Matching*, The Univ. of Arizona Tech. Rep. 86-1, Jan. 1986.
2. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
3. R. E. Griswold, *The Construction of Variant Translators for Icon*, The Univ. of Arizona Tech. Rep. 83-19a, June 1984.