A debugging environment for functional programming in CENTAUR

Samuel Kamin

Rapports de Recherche

N° 1265

Programme 1
Programmation, Calcul Symbolique et Intelligence Artificielle

A DEBUGGING ENVIRONMENT FOR FUNCTIONAL PROGRAMMING IN CENTAUR

Samuel KAMIN

Juillet 1990

A debugging environment for functional programming in Centaur

Un environment de debugging pour la programmation fonctionnelle en Centaur

Samuel Kamin
INRIA Sophia-Antipolis
June 18, 1990

Abstract

We present a trace-based debugging environment for a lazy, functional language. We argue that traces are a natural, even inevitable, approach to debugging of lazy languages, because stop-and-examine techniques run up against the unpredictability of lazy evaluation. We give a formal definition of *trace*, describe how the **Centaur** system was used to build the environment, and show our system being used to debug a small program.

The more general goal of this work is to demonstrate a "hypertextual" approach to trace-based debugging. Our argument is that using hypertext techniques overcomes one of the most serious problems traditionally associated with traces: information overload.

Keywords: Functional programming, Debugging, Centaur

Résumé

Nous présentons un environnement de debugging basé sur des traces pour un langage fonctionnel à évaluation paresseuse. Nous soutenons que les traces sont une méthode tout à fait naturelle — même incontournable — pour la mise au point dans de tels langages, parce que l'aspect chaotique de l'évaluation dans l'ordre normal rend inefficaces les méthodes du type "arrêt-et-inspection." Nous présentons une définition formelle de notre notion de trace, puis nous décrivons l'utilisation du système **Centaur** pour construire l'environnement, et nous démontrons l'utilisation de notre système pour mettre au point un petit programmme.

Le but général de ce travail est d'offrir une approche "hypertext" pour la mise au point à l'aide de traces. Nous croyons que les méthodes hypertext permettront de maîtriser le problème le plus grave lié aux traces: l'excès d'information.

Mots Clés: Programmation fonctionelle, Debugging, Centaur

A debugging environment for functional programming in Centaur *

Samuel Kamin

INRIA Sophia-Antipolis
Route des Lucioles
06560 Valbonne
France

June 1, 1990

Abstract

We present a trace-based debugging environment for a lazy, functional language. We argue that traces are a natural, even inevitable, approach to debugging of lazy languages, because stop-and-examine techniques run up against the unpredictability of lazy evaluation. We give a formal definition of *trace*, describe how the CENTAUR system was used to build the environment, and show our system being used to debug a small program.

The more general goal of this work is to demonstrate a "hypertextual" approach to trace-based debugging. Our argument is that using hypertext techniques overcomes one of the most serious problems traditionally associated with traces: information overload.

Contents

*This work has been supported in part by AT&T under the Illinois Software Engineering Project. Author's current address is Computer Science Dept., 1304 W. Springfield, University of Illinois, Urbana, IL 61801, and his electronic mail address is kamin@cs.uiuc.edu.

1 Introduction

The difficulty of debugging programs in lazy functional languages has often been noted. The fundamental problems are:

Visibility. The debugging process in conventional languages involves a careful dissection of the state of the computation at selected points. In those languages, most of the state can be observed quite directly: only a small part of it is hidden as a result of scope rules. An important point to stress, however, is that the visibility boundaries created by scope rules can be a hindrance in debugging, even when they are otherwise perfectly appropriate. Consider for example the debugging of a recursive procedure in a conventional language. When the procedure is halted during a recursive call, only the latest instances of its local variables and formal parameters are visible. Yet it is quite natural for the programmer to want to see other instances of these variables, such as their values in the previous call or in the earliest call of this procedure.

Lack of visibility becomes a particular problem in languages in which scope rules are heavily used in support of high-level abstractions, such as object-oriented and functional languages. In functional languages, much of the state is hidden in closures and suspensions, and few debugging environments provide access to these.

Predictability. The discussion of "state" above assumes a predictable order of evaluation in which the programmer can understand how the computation moves from one state to another (and thereby compare it to his understanding of how it *should* move). Lacking this kind of predictability, the most common methods of debugging — inserting print statements and single-stepping in a debugging system — are not applicable.

Lazy languages have a deterministic evaluation order. Moreover, that order is not difficult to understand *in theory*. However, experience shows that applying the theory in specific cases is extremely difficult. Thus, from a debugging point of view, the execution order is effectively unpredictable.

These issues point to fundamental difficulties in debugging lazy functional languages, yet the situation is not entirely bleak. Consider the problem of unpredictability. Even though the step-by-step computation of a lazy program may be impenetrably complex, the *overall* evaluation, regarded as a tree, has a very simple structure. This suggests that *execution traces* may be a useful tool, as indeed has been suggested in [9]. The debugging environment presented here is based on the use of traces.

The problem with traces, as is well known from experience with sequential languages, is that they are huge. The traditional approach to their use has been, therefore, to run the program, storing the trace in a file, then print the trace, thereby producing reams of output containing a very small amount of interesting information. Clearly, single-stepping, when applicable, is a far more efficient and precise method of debugging.

The visibility issue has a similar characteristic: the state is a large structure only a small part of which may be interesting at any given time.

Thus, the question becomes: how can we provide a method of interactively navigating over large structures? Once we have a facility for doing so, the use of traces becomes feasible, as it will allow the programmer to home in on the interesting part of the trace, much as single-stepping tools allow him to home in on the interesting execution steps. Access to the whole state can be provided in the same way. It is precisely its support for display and navigation over large structures that makes CENTAUR an appropriate system for developing our debugging environment.

CENTAUR [1, 2] is a generic interactive programming environment. A programming environment for a specific language is constructed by giving formal syntactic *and semantic* language definitions. A large library of LISP functions is provided with which the language designer can further customize the programming environment.

The "large structures" with which CENTAUR works are *abstract syntax trees* (AST's), a concept supported in the system by the Virtual Tree Processor (VTP). The VTP consists of a set of LISP functions with which to describe an abstract syntax and then build trees conforming to it; it allows as well for attaching attributes to AST nodes, limited tree pattern-matching, persistent storage of AST's, and other miscellaneous services.

The abstract syntax of a language is the central part of its definition. A tool called METAL is used to describe a language's concrete syntax and the translation of concrete syntax trees to AST's. Another tool, PPML, is used to describe the translation of AST's to character streams — i.e., the *pretty-printing* of AST's. When an AST is pretty-printed, the system maintains a map associating regions of the displayed text with AST nodes, so that a user can click on the text to select an AST node. The selection thus made can be used in various ways: it can be clipped or copied, copied into another window, have its display changed, etc.

VTP trees are not used only for AST's but for all data that is to be displayed to the user. In particular, the result of a computation is a VTP tree. Thus, PPML can be used to display the results of computations and, if those results are too large to fit in the available window, to navigate over them.

Here, then, is how we use Centaur: The semantics of a program in our lazy functional languages gives, as the "value" of a program, a trace. This is nothing but a (large!) VTP tree. PPML is used to pretty-print it and allow for interactive navigation. Finally, some Lisp code is added to ameliorate the programming environment by, for example, allowing for opening multiple windows on the trace.

Our presentation begins with a review of past work in this area and a more careful look at our language and its trace semantics. We then give a further overview of Centaur, including parts of the language definition. The major section describes and exemplifies the experimental programming environment we have built.

2 Previous work

Two recent papers addressing the problem of debugging in lazy languages are O'Donnell and Hall [9] and Toyn and Runciman [11]. O'Donnell and Hall [9] try several approaches, culminating in an interactive system in which the user can enter the body of a function and request the evaluation of subexpressions occurring there. Along the way, they consider the idea of transforming a program to produce a trace as well as a value, but reject it for technical reasons (the transformation is not semantics-preserving because the printing of the trace information is an eager process). They do not consider the visibility problem, and all of their examples in fact involve *non-lazy* evaluation — that is, the computations are dominated by strict primitive operations. The main emphasis in their work is on developing the debugging system *within* the lazy language itself.

Toyn and Runciman's work [11] is based on computation *snapshots*, a snapshot being a picture of the computation's state at any chosen moment. (This is not the same as traces, which are much larger.) The problem they consider is how to modify combinator-reduction code so that enough information is retained during the computation to enable the snapshot to be presented in a readable form at any time.

Our work differs from these in several ways:

- We base our debugging system on traces. In contrast to [9], this entails no danger of fortuitous over-eagerness because we accomplish it by changing the semantics of the language rather than by transforming programs.
- We provide a formal definition of *trace*.
- Our principal concern is with presenting the trace to the user in a usable form and providing tools to navigate over it. Toyn and Runciman set the stage for the current work when they say [11, pg. 360]: "A more sophisticated snapshot formatter might act as an interactive browser, unfolding and folding parts of a snapshot on command from the user."
- We have made no attempt to compute the trace of a computation as a byproduct of normal evaluation, as is the main contribution of [11], nor to reduce the memory usage entailed by the construction of traces. Thus, our debugging environment cannot at present be considered a realistic system, but rather an experiment in structuring such an environment.

3 What is a trace?

3.1 The language

The language we implement is intended to be a minimal functional language with lazy semantics. It is drawn from Kamin [7, Chapter 5]; as there, it is called Sasl in recognition of the Sasl language of David Turner [12], though it shares only its most basic features: being functional and lazy. Its syntax is given in Figure 1, and Figure 2 is an example, the `first-n-primes` function.

We assume the semantics of the language is understood intuitively: it is really just λ-calculus with some primitive data types and operations added. In order precisely to define *trace*, we first give an operational semantics of the language; the 0 subscript is used to distinguish this from the trace semantics to follow:

```
input        ⟶  expression | ( set variable expression )
expression   ⟶  value
              |  variable
              |  ( if expression expression expression )
              |  ( expression+ )
value        ⟶  integer | quoted-const | ( lambda arglist expression ) | value-op
arglist      ⟶  ( variable* )
value-op     ⟶  + | - | = | < | > | cons | car | cdr | null? | ···
```

Figure 1: Syntax of SASL

```
(set first-n (lambda (n l)
       (if (null? l) '()
          (if (= n 0) '()
             (cons (car l) (first-n (- n 1) (cdr l)))))))

(set ints-from (lambda (i) (cons i (ints-from (+1 i)))))

(set remove-multiples (lambda (n l)
       (if (divides n (car l))
          (remove-multiples n (cdr l))
          (cons (car l) (remove-multiples n (cdr l))))))

(set sieve (lambda (l)
       (cons (car l) (sieve (remove-multiples (car l) (cdr l))))))

(set primes (sieve (ints-from 2)))

(set first-n-primes (lambda (n) (first-n n primes)))
```

Figure 2: Function `first-n-primes`

$Expression = Integer + Symbol + ListConst + Variable + Application + Abstraction$

$Value_0 = Integer + Symbol + List_0 + Closure_0 + Thunk_0 + Primitive_0$
$List_0 = Value_0 \times Value_0 + \{\text{nil}\}$
$Closure_0 = Expression \times Environment_0$
$Thunk_0 = Expression \times Environment_0$
$Primitive_0 = Value_0^* \longrightarrow Value_0$
$Environment_0 = Variable \longrightarrow Value_0$

The closure (resp. thunk) containing expression $\mathtt{e}$ and environment ρ will be denoted $\langle\!\langle \mathtt{e},\ \rho \rangle\!\rangle$ (resp. $\lhd \mathtt{e},\ \rho \rhd$).

```
$eval_0 : Expression \times Environment_0 \longrightarrow Value_0$
$\mathtt{e}, \rho \longmapsto$ case type($\mathtt{e}$) of
        constant (integer, symbol, or list) : $\mathtt{e}$
        lambda expression : $\langle\!\langle \mathtt{e},\ \rho \rangle\!\rangle$
        variable : let $v = \rho(\mathtt{e})$ in if $v = \lhd \mathtt{e}',\ \rho' \rhd$ then $eval_0(\mathtt{e}',\ \rho')$ else $v$
        application ($\mathtt{e}_0\ \mathtt{e}_1 \ldots \mathtt{e}_n$) :
                let $v_0 = eval_0(\mathtt{e}_0,\ \rho), v_1 = \lhd \mathtt{e}_1,\ \rho \rhd, \ldots, v_n = \lhd \mathtt{e}_n,\ \rho \rhd$
                in if $v_0 = \langle\!\langle (\mathtt{lambda}\ (\bar{x})\ \mathtt{e}'),\ \rho' \rangle\!\rangle$
                   then $eval_0(\mathtt{e}',\ \rho'[v_1/x_1,\ \ldots,\ v_n/x_n])$
                   else $v_0(v_1,\ \ldots,\ v_n)$
```

In the last line, v_0 is a value of type $Primitive_0$.

This semantics ignores the presence of a global environment. This global environment stores the values of variables defined by top-level `set`'s, and also contains the values of variables that denote primitive functions, such as `+` and `if`. We give the definitions of several of these functions:

$$+ : \lhd \mathtt{e}_1,\ \rho_1 \rhd,\ \lhd \mathtt{e}_2,\ \rho_2 \rhd \ \longmapsto\ eval_0(\mathtt{e}_1,\ \rho_1) + eval_0(\mathtt{e}_2,\ \rho_2)$$

```
if : $\lhd e_1,\ \rho_1 \rhd,\ \lhd e_2,\ \rho_2 \rhd,\ \lhd e_3,\ \rho_3 \rhd$
        $\mapsto$ if $eval_0(e_1,\ \rho_1) \neq$ nil then $eval_0(e_2,\ \rho_2)$ else $eval_0(e_3,\ \rho_3)$
cons : $x, y \mapsto < x, y >$
car : $\lhd e,\ \rho \rhd \ \mapsto$ let $< x, y > = eval_0(e,\ \rho)$
                        in if $x = \lhd e',\ \rho' \rhd$ then $eval_0(e',\ \rho')$ else $x$
```

As a meta-evaluation rule, we assert that whenever a thunk is evaluated, it is replaced in memory by its value, so that all environments or lists retaining a reference to the thunk get the benefit of its having been evaluated. Therefore, the arguments to primitive operations like + may not be thunks; in that case, just bypass *eval*$_0$ and do the operation. We ignore for now the problem of errors such as an attempt to add non-numbers.

3.2 Trace semantics

In our trace semantics, the value of an expression is to be a history of the evaluation "steps" of *eval*$_0$. This entails a simple modification of the operational semantics in which the evaluation steps become a part of the value.

$XValue = Value \times XValue^* \times XValue^* + Thunk$
$Value = Integer + Symbol + List + Closure + Primitive$
$List = XValue \times XValue + \{\text{nil}\}$
$Closure = Expression \times Environment$
$Thunk = Expression \times Environment$
$Primitive = XValue^* \rightarrow Value$
$Environment = Variable \rightarrow Value$

The domain *Expression* is unchanged.

$eval : Expression \times Environment \rightarrow XValue$

The components of a non-thunk *XValue* need some explanation. The first is just the "simple value" obtained from the evaluation. If π_1 is the first projection of a tuple, then we could say (ignoring lists, for which a more complicated, but intuitively similar, equivalence holds), for all e and ρ:

$$\pi_1(eval(e,\ \hat{\pi}_1(\rho))) = eval_0(e,\ \rho),$$

where $\hat{\pi}_1(\rho) = \lambda x.\pi_1(\rho(x))$. The second component of an *XValue* is the list of traces of all subexpressions of an application. The last component is an empty sequence unless the value was obtained as an application of a closure; in that case, it contains the trace of the evaluation of the closure's body (thus, it contains at most one item).

The definition of *List* is also noteworthy. Since lists are included in the *Value* domain rather than the *XValue* domain, it might seem more natural for them to be pairs of *Value*'s. However, this would cause a problem: Consider the expression (cons e$_1$ e$_2$). It results in the *XValue*: $<?, < \lhd e_1,\ \rho \rhd,\ \lhd e_2,\ \rho \rhd >, <>>$. If we used pairs of *Value*'s, the "?" would eventually be the pair

$$< \pi_1(eval(e_1,\ \rho)),\ \pi_1(eval(e_2,\ \rho)) >,$$

once e$_1$ and e$_2$ were evaluated (if ever), but what would it be when first created? It would have to be a pair the first component of which gets overwritten with $\pi_1(eval(e_1,\ \rho))$ once e$_1$ is evaluated. However, our meta-evaluation rule only allows for thunks to be overwritten by values produced by *eval*, that is, by *XValue*'s. Thus, the meta-evaluation rule would have to be extended in what turns out to be a rather complex way. To avoid this, we use pairs of *XValue*'s.

```
$eval : Expression \times Environment \rightarrow XValue$
$e,\ \rho \ \mapsto$ case type(e) of
    constant : $< e, <>, <>>$
    lambda expression : $< \langle\langle e,\ \rho \rangle\rangle, <>, <>>$
    variable : let $v = \rho(x)$ in if $v = \lhd e',\ \rho' \rhd$ then $eval(e',\ \rho')$ else $v$
    application (e$_0$ e$_1 \ldots$ e$_n$) :
            let $v_0 = eval(e,\ \rho), v_1 = \lhd e_1,\ \rho \rhd,\ \ldots,\ v_n = \lhd e_n,\ \rho \rhd$
            in if $\pi_1(v_0) = \langle\langle$(lambda ($\vec{x}$) e'), $\rho' \rangle\rangle$
```

```
then let $v' = eval(e', \rho'[v_1/x_1, \ldots, v_n/x_n])$
     in $< \pi_1(v'), < v_0, v_1, \ldots, v_n >, < v' >>$
else let $v' = \pi_1(v_0)(v_1, \ldots, v_n)$
     in $< v', < v_0, v_1, \ldots, v_n >, <>>$
```

The primitives are defined as follows; note that they define functions from *XValue*'s to *Value*'s:

```
+ : $\triangleleft e_1, \rho_1 \triangleright, \triangleleft e_2, \rho_2 \triangleright \mapsto \pi_1(eval(e_1, \rho_1)) + \pi_1(eval(e_2, \rho_2))$
if : $\triangleleft e_1, \rho_1 \triangleright, \triangleleft e_2, \rho_2 \triangleright, \triangleleft e_3, \rho_3 \triangleright$
        $\mapsto$ if $\pi_1(eval(e_1, \rho_1)) \neq$ nil then $\pi_1(eval(e_2, \rho_2))$ else $\pi_1(eval(e_3, \rho_3))$
cons : $x, y \mapsto < x, y >$
car : $\triangleleft e, \rho \triangleright \mapsto$ let $< x, y > = eval(e, \rho)$
                 in if $x = \triangleleft e', \rho' \triangleright$ then $\pi_1(eval(e', \rho'))$ else $\pi_1(x)$
```

The meta-evaluation rule about replacing thunks by their values (that is, their *XValue*'s) still holds.

Thus, in general the trace may be regarded as a tree, each node representing the evaluation of an expression, and its children the trees of its subexpressions, plus, if the expression is the application of a closure, a child giving the trace of the body of the closure. We think of this last subtree as the "control" component of the trace of the node.

This tree is not very different from what would be obtained if we considered an eager evaluation language. The crucial difference is that in the eager case, the tree grows in a simple, predictable order (namely: pre-order), which is why single-stepping through the execution makes sense. Here, most nodes of the tree *start out* as thunks and are later *expanded* on demand. Thus, the tree grows in a far more complex and surprising way and evaluation is difficult to follow step by step. On the other hand, these traces are huge. Thus, we view our task as: (1) constructing a language processor that produces *XValue*'s, and then (2) providing a "hypertext" interface for the exploration of these *XValue*'s.

4 Overview of CENTAUR

CENTAUR is a system for defining "formalisms," that is, languages. It allows for the construction of high-quality user-interfaces for these formalisms. The best-known examples of formalisms are programming languages, where the semantic definitions are interpreters and compilers; another class of formalisms are those representing proofs in formal logics.

It is not misleading to identify "formalism" with "abstract syntax." Given a formalism, some tools that help make it useful are:

- Translator from concrete to abstract syntax (a *parser*). A parser would read a file containing the characters "`(set x (cons 1 '()))`" and produce the abstract syntax tree (AST):

```
          definput
         /        \
    var(x)       applic
                /      \
            cons        exp_s
                       /     \
              integer(1)    const_s[]
```

  The CENTAUR utility METAL is used to create parsers.[1]

- Translator from abstract to concrete syntax (a *pretty-printer*). PPML is the program that creates pretty-printers.

- Translators from abstract syntax to the same or a different abstract syntax. This includes program transformers (from one formalism to itself), interpreters (from the formalism of source programs to the formalism of results), and compilers (formalism of source programs to formalism of object programs). TYPOL can be used to define such computations.

[1] There is also a newer utility, SDF, for the same purpose, but it has not been used in this project and will not be further discussed.

Thus, the utilities in CENTAUR are used to define new formalisms and the tools that make them useful.

4.1 CENTAUR tools

4.1.1 The VTP

The Virtual Tree Processor [8], or VTP, is the heart of the system. It is a collection of LISP functions with which formalisms can be defined and AST's in those formalisms constructed. VTP is, in effect, a data base system for trees, handling not only their construction but also their storage on disk.

The functions of VTP are almost never used directly by the user. Instead, tools are provided, as described above and in more detail below, to allow the specification of VTP actions at a higher level. Thus, VTP trees are created either by parsing concrete syntax (using a METAL-created parser) or by doing some computation (using a TYPOL program).

Note that VTP is used both to create formalisms, i.e. abstract syntaxes (using the **{formalism}:make** function), and to construct AST's in those formalisms (using the **{tree}:make** function).

4.1.2 Specifying abstract and concrete syntax in METAL

METAL [6] is used to describe both a formalism — i.e. abstract syntax — and a corresponding concrete syntax. Thus, a METAL specification is divided in two parts. METAL processes the abstract syntax part by calling the appropriate VTP operations (in particular, **{formalism}:make**) to create the formalism. The concrete syntax part of the specification is used to create a parser which, when applied to a stream of characters, will create the corresponding AST (by calling **{tree}:make**).

Figure 3 contains part of the METAL specification for our language. The top half is the abstract syntax, which itself is in two parts: The list of operators, giving the phyla of each operator's arguments; and the list of phyla, giving all the operators in each phylum. Thus,

```
def -> VAR EXP
```

says that a tree whose root is labeled **definput** must have two children which are in phyla **VAR** and **EXP**, respectively.

```
INPUT ::= definput expinput;
```

says that trees labeled **definput** or **expinput** are in the phylum **INPUT**.

The concrete syntax is defined in the usual way and, in fact, the standard UNIX[2] utility **yacc** is used to generate the parser; that is, METAL produces, among other things, a **yacc** input file. Following each concrete syntax rule is an expression computing an AST. For example, consider:

```
<def> ::= #( set <var> <exp> #) ;
    def ( <var>, <exp> )
```

The first line is a BNF rule. The second says that the AST corresponding to this concrete syntax is:

$$\begin{array}{ccc} & \texttt{def} & \\ / & & \backslash \\ t_1 & & t_2 \end{array}$$

where t_1 and t_2 are the AST's for **<var>** and **<exp>**, respectively.

In summary, a METAL specification contains a description of a formalism, giving both its abstract and concrete syntax. When METAL is run, it creates the formalism using VTP and also produces a **yacc** input file. The user compiles the **yacc**. After loading the compiled file, CENTAUR has the capability to read files of concrete syntax, parse them, and translate them to VTP trees.

4.1.3 Pretty-printing via PPML

With the METAL specification, we are in a position to read a file of characters and produce a VTP tree. However, there is no way to display the tree. PPML is a tool for describing the textual layout of a VTP tree.

Our debugging environment uses two PPML files, one describing the display of SASL programs and the other the display of traces. Figure 4 contains parts of the former. For example, the rule:

[2]UNIX is a trademark of AT&T.

```
definition of SASL is
   abstract syntax
      input_s -> INPUT * ...;
      definput -> DEF;
      expinput -> EXP;
      def -> VAR EXP;
      var_s -> VAR * ...;
      exp_s -> EXP * ...;
      var -> implemented as IDENTIFIER;
      intconst -> implemented as INTEGER;
      symconst -> implemented as IDENTIFIER;
      const_s -> CONST * ...;
      value_s -> VALUE * ...;
      lambda -> VAR_S EXP;
      applic -> EXP EXP_S;
         ⋮
      CONST ::= intconst symconst const_s;
      EXP ::= applic CONST VAR lambda;
      VAR ::= var;
      EXP_S ::= exp_s;
      VAR_S ::= var_s;
      PROG ::= program;
      INPUT_S ::= input_s;
      INPUT ::= definput expinput;
      DEF ::= def;
         ⋮
   rules
      <input_s> ::= <input>;
         input_s-list (( <input> ))
      <input_s> ::= <input_s> <input>;
         input_s-post (<input_s>, <input>)
      <input> ::= <def>;
         definput(<def>)
      <input> ::= <exp>;
         expinput(<exp>)
      <def> ::= #( set <var> <exp> #) ;
         def ( <var>, <exp> )
      <exp> ::= #( lambda <arglist> <exp> #) ;
         lambda( <arglist>, <exp> )
      <exp> ::= #( <exp> <exp_s> #) ;
         applic ( <exp>, <exp_s> )
         ⋮
end definition
```

Figure 3: Fragment of the METAL specification of SASL

```
def(*v,*e) -> [<hv 0, bigtab, 0> [<h 0> "(set " *v] " " [<h 0> *e ")"]];
lambda (*vars, *exp)
   -> [<hv 0, tab, 0> [<h 0> "(lambda " *vars] " " [<h 0> *exp ")"]];
var_s () -> "()";
var_s (*v, **vs) -> [<h 0> "(" *v (" " **vs) ")"];
applic (*op, *exps) -> [<h 0> "(" *op <h 0> " " *exps ")"];
exp_s () -> ;
exp_s (*exp, **exps) -> [<hv 0, tab, 0> *exp (" " **exps)];
```

Figure 4: Fragment of PPML specification of SASL

```
def(*v,*e) -> [<hv 0, bigtab, 0> [<h 0> "(set " *v] " " [<h 0> *e ")"]];
```

says that an AST node with operator **def** (which represents a top-level definition) is to be displayed by displaying the characters "**(set**", then the first subtree (the variable), a space, the second subtree (the expression) and then a closing parenthesis. Furthermore, these are all to be placed on a single line, if possible; if not, they should be split between the variable and the expression, with a large tab on the second line.

Another feature of PPML that shows up in Figure 4 needs to be explained. In the second rule for displaying lists of variables (var_s), we find a variable **vs, which actually represents a *list* of variables. When such a list variable is placed in parentheses on the right-hand-side of a rule, there is an implicit iteration over all elements of the list. Thus, in that rule, the "(" " **vs)" is equivalent to "" " $**vs_1$ " " ... " " $**vs_n$", if **vs has n elements.

4.1.4 Semantic definitions in TYPOL

The semantics of a language is defined using the method of natural semantics [5, 10], as implemented in the TYPOL system [3]. The input to TYPOL is a set of files containing logical rules of inference. Originally intended to define type-checking and other context-sensitive syntax of programs, TYPOL can in fact be used to define arbitrary computations on programs. A typical rule is the one defining the semantics of a while statement in a simple programming language:

```
BOOL_EVAL(env |- b -> true, env')
& env' |- S -> env''
& env'' |- while(b, S) -> env'''
---------------------------------
env |- while(b, S) -> env'''
```

This says the judgement below the horizontal line is valid if the three judgements above it are. Specifically, it says that the **while** statement, if started in state **env**, will results in state **env'''** if: **b** evaluates to true, while possibly changing the state to **env'** as a result of side effects; **S** then leads from **env'** to **env''**; and then continuing the **while** eventually leads to **env'''**.

By using primitives **gettree** and **sendtree**, a TYPOL program can grab a VTP tree and return one as a result. Thus, the first rule in our SASL semantics is:

```
BASE_ENV(rho)
& rho |- p, lisp "lisp" -> xl, rho'
---------------
() ;
    provided gettree("k", subject, p);
    do sendtree("return-values", xl) &
       sendtree("global-env", rho');
```

The judgement "()" is just a dummy used to start evaluation. **BASE_ENV** is a predicate defined elsewhere which guarantees **rho** will contain the bindings of primitive operations. The main part of this rule is the second antecedent, which says: starting in environment **rho**, program **p** will result in the list **xl** of *XValue*'s and global environment **rho'**. (The "**lisp "lisp"**" part gives the path to the beginning of the source program, but this technicality is not of interest here.)

"**provided**" and "**do**" allow actions to be taken before and after a rule is used. The **gettree** says the TYPOL variable **p** is obtained from the VTP tree "**k**", and the **sendtree** assigns the resulting value of TYPOL variable **xl** to the VTP tree **return-values** and the value of **rho'** to **global-env**. The main point here is that the computation defined by this TYPOL specification is a function whose input is a VTP tree and whose output is two VTP trees.

4.1.5 CENTAUR user-interface tools

A great deal of effort has been invested in giving CENTAUR a user-interface that is up-to-date and easy to use as well as customizable.

Given a VTP tree and a pretty-printer (i.e. PPML program), the display is placed in a special type of window called a *ctedit*. A ctedit allows for structural pointing and editing; that is, a subtree can be selected by clicking with the mouse, and the selected subtree can then be clipped, copied, and so on. It

```
xvalue -> VALUE X_VALUE_S X_VALUE_S;
xvalue_s -> X_VALUE * ...;
mapsto -> VAR X_VALUE;
envcons -> MAPSTO ENV;
envnil -> implemented as SINGLETON;
thunk -> EXP PATH ENV X_VALUE;
envintro -> ENV PATH X_VALUE;
pair -> X_VALUE X_VALUE;
gate -> implemented as TREE;
PATH ::= gate;
VALUE ::= error intconst symconst pair nil closure
          BINARYOP UNARYOP TERNARYOP;
VALUE_S ::= value_s;
X_VALUE ::= xvalue thunk envintro;
X_VALUE_S ::= xvalue_s;
MAPSTO ::= mapsto;
ENV ::= envcons envnil diffenv;
```

Figure 5: Abstract syntax of traces

also allows for varying certain physical aspects of the window, such as pulldown and popup windows and highlighting.

Connections can be made between trees so that clicking in one ctedit causes highlighting in another. This is used, for example, in the system for reporting errors in METAL specifications: a window containing the list of errors is created and clicking on an error causes the source of that error, in the window containing the METAL specification, to be highlighted.

4.2 How we use CENTAUR

The idea of our system is to compute the trace of a computation as the value of that computation. Thus, the execution of a program results in the creation of a large VTP tree which is displayed via its own PPML. The structure of this trace has already been described in section 3.2. To make it concrete, we give in Figure 5 the abstract syntax of traces. This is actually a part of the METAL specification that was not shown in Figure 3. A trace is in phylum `X_VALUE` and can be one of three things: an `xvalue`, a `thunk`, or an `envintro`. The first two are just as we described in section 3.2; the `envintro` is a way of attaching both environments and pointers into source code (the `PATH` component) to the trace.

Our TYPOL code has as its input a VTP tree of phylum `INPUT_S`, and produces two VTP trees, as described in section 4.1.4: `return-values` of type `X_VALUE_S` and `global-env` of type `ENV`.

4.2.1 Displaying traces

The user's view of the debugging process is that values appear on the display — at first, these are just the values of top-level expressions — and the user may ask a value to "explain itself." She does this by selecting a value (clicking on it) and then, by selecting a menu item, asking for some information about how that value came to be. There are three kinds of information that can be provided for each value:

1. The expression whose evaluation produced the value.

2. The environment in which the expression was evaluated.

3. The "history" of the value, meaning the sequence of function applications that led to it.

In addition, for closure values there are two other items:

5. The `lambda` expression contained in the closure.

6. The environment contained in the closure. (N.B. This is not the same as (2) above.)

This is too much information to present in a single display, so the following decisions were made:

- For (1) and (4), where an expression from the original program is to be displayed, that expression is highlighted in the source window. This is, in any case, preferable to displaying a separate copy of the expression, since it gives the user more context.
- Environments are displayed by opening new windows for them; this is how we handle (2) and (5).

The main window is the one in which the values of top-level expressions in the user's program are originally displayed (as simple values), and it is here that the histories of these values are presented, at the user's request.[3] We would now like to explain what these history displays contain, how the user controls them, and how the PPML specification implements them.

First, what is in the display? Assume a generic *XValue*

$$v = < s, < v_0, v_1, \ldots, v_n >, < v_{\text{next}} >> .$$

At the top level, the user may wish to see only the simple value of v, namely s (though note that if this is a list, producing its display as a simple value involves traversing all the *XValue*'s it contains). If, however, the full history is requested, it would be displayed according to scheme $\mathcal{F}$:[4]

$$\mathcal{F}(v) = \boxed{\begin{array}{l} s \\ \mathcal{H}(v) \end{array}}$$

$$\begin{aligned} \mathcal{H}(v) &= \text{empty, if } n = 0 \\ &\boxed{\begin{array}{l} = (v_0\ v_1\ \ldots\ v_n) \\ \quad \mathcal{F}'(v_0) \\ \quad\quad \vdots \\ \quad \mathcal{F}'(v_n) \\ \mathcal{H}(v_{\text{next}}) \end{array}}, \text{ otherwise} \end{aligned}$$

$$\begin{aligned} \mathcal{F}'(v) &= \text{empty, if } n = 0 \\ &\quad \mathcal{F}(v), \text{ otherwise} \end{aligned}$$

Well, not really. The display given by scheme $\mathcal{F}$ is much too big, in general, and can't be fully displayed; that would amount to giving the entire history of the computation up to this point. Aside from the original choice of whether to display the history of the top-level value, the user can control the display in one way: by deciding which of the v_i to display with the $\mathcal{F}'$ format within the $\mathcal{H}$ format.

To illustrate, suppose the input program is:

```
(set f (lambda (x) (+ (* x x) 1)))
(f 3)
```

The original display of `(f 3)` is its simple value:

```
10
```

By clicking on the 10 and requesting the history, this is displayed in the $\mathcal{F}$ format, which is:

```
10
= (<<->> 3)
= (+ 9 1)
```

"<<->>" represents a closure value. Here, none of the values in the applications is itself displayed with $\mathcal{F}'$ scheme. Clicking on the 9 and requesting its history gives:

```
10
= (<<->> 3)
= (+ 9 1)
    9
    = (* 3 3)
```

[3] The following discussion on the display of histories also applies to environment windows.

[4] This description gives the actual physical layout of the display. $\mathcal{F}(v)$, for example, consists of the display of s stacked vertically, with no indent, above the $\mathcal{H}(v)$ display. The outline boxes are not part of the display.

This is, in fact, the full display for value 10, since none of the other values in the applications has any history.

We next consider how to produce this display in PPML, and how to achieve user control over it. We have a separate PPML specification for the display of traces (including **X_VALUE**'s and **ENV**'s). Though the entire PPML specification contains 97 rules, the essential rules expressing the recursive structure of the trace display can be readily explained, based on the $\mathcal{F}$ and $\mathcal{H}$ formats. First, it is necessary to understand that a PPML specifcation is divided into chapters which correspond to different pretty-printing methods, or *contexts*. If we wish tree `t` to be pretty-printed in context `C`, we write `C::t`. The major contexts in this specification are:

top level: this context is unnamed. It allows for generic *XValue* v to be displayed in one of two ways: $\boxed{s}$, or $\boxed{\begin{array}{l} s \\ \texttt{HIST}(v) \end{array}}$; the latter corresponds to format $\mathcal{F}$.

SV: Prints the simple value of an **xvalue**, i.e. its first component. (In the case of lists, this involves a traversal of a tree of **X_VALUE**'s.)

HIST: Displays the history of a value by showing its simple expression, recursively showing its subexpressions, and showing the history of the "next expression," i.e. the third component of the **xvalue**. This corresponds to the $\mathcal{H}$ scheme: $\boxed{\begin{array}{l} \texttt{SE}(v) \\ \quad \texttt{SUBEXP}(v_0) \\ \quad \vdots \\ \quad \texttt{SUBEXP}(v_n) \\ \texttt{HIST}(v_{\text{next}}) \end{array}}$.

SE: Prints the "simple expression" associated with an **xvalue**, which is just the list of simple values of the subexpressions (second component of the **xvalue**). These are surrounded by parentheses, to indicate that they represent an application: $\boxed{\texttt{= (SV}(v_0)\ \texttt{SV}(v_1)\ \ldots\ \texttt{SV}(v_n)\texttt{)}}$.

SUBEXP: At the user's discretion, this is either empty, or is the same as the second top level format; in other words, it is $\mathcal{F}'$.

To give an example, the central rule of the **HIST** context is:

```
*x where *x in xvalue (*v, xvalue_s(**sel), *h) ->
      [<v 0, 0>
         [<hv 1, tab, 0> SE::*x]
         [<h 0> " " [<v 0, 0> (SUBEXP::**sel)]]
         [<h 0> HIST::*h!+1]];
```

Three technical points: (1) The left-hand side "`*x where *x in` *p*," where *p* is any pattern, is the same as *p* alone, except that the entire tree has a name that can be used in the right-hand side. Of course, this feature is only useful if the tree is to be displayed in a different context. (This feature is similar to "as" patterns in functional languages like Haskell [4].) (2) The specification "`[<v 0, 0> (SUBEXP::**sel)]`" displays each of the items in the list `**sel` in the **SUBEXP** context, broken vertically. (3) The notation "`!+1`" on the last line ensures that each history is displayed as fully as the first; other items (such as long lists) may be holophrasted, but not histories. Having noted these facts, we can see that this rule implements the $\mathcal{H}$ format directly.

There are two places at which the user's control over the display is asserted: At the top level, the user chooses between the full display and the simple value; and in the **SUBEXP** context, she chooses between the full display and nothing. To communicate the user's request to the pretty-printer, the VTP/PPML *annotation* facility is used. The VTP provides functions with which named annotations can be placed into AST nodes. The Lisp code that is invoked when the users clicks on a value and selects the "`Show history`" menu item places an annotation named "`show-hist`" on a node. PPML can check whether the "`show-hist`" annotation is present at a node by adding "`^show-hist`" to the left-hand side of a rule. Thus, we find these two rules at the top level:

```
*x where *x in xvalue (*v, *se, *h) ^show-hist ->
      [<v 0, 0>
         [<h> SV::*x]
         [<h 0> HIST::*x]];
x where *x in xvalue (*v, *se, *h) -> [<h> SV::*x];
```

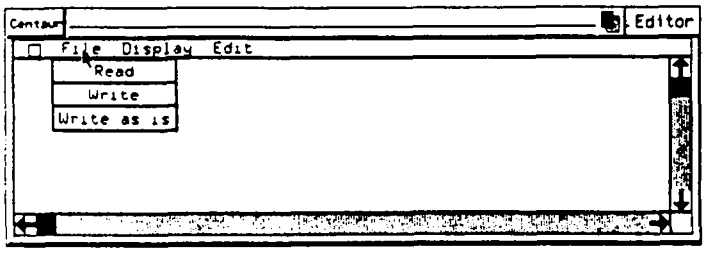
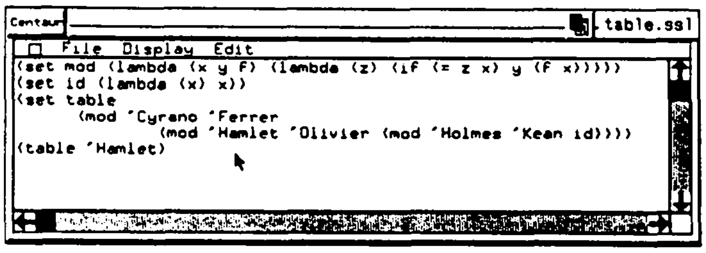


Figure 6: Before reading source file



```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
      (mod 'Cyrano 'Ferrer
            (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

Figure 7: After reading source file

The first rule says to display an **xvalue** by giving its simple value and its history, the second to display only its simple value. These very similar rules appear in the **SUBEXP** chapter:

```
*x where *x in
    xvalue (*v, *se, *h) ^show-hist ->
       [<v 0, 0>
          [<h> SV::*x]
          [<h 0> HIST::*x]];
xvalue (*v, *se, *h) -> ;
```

5 The debugging environment

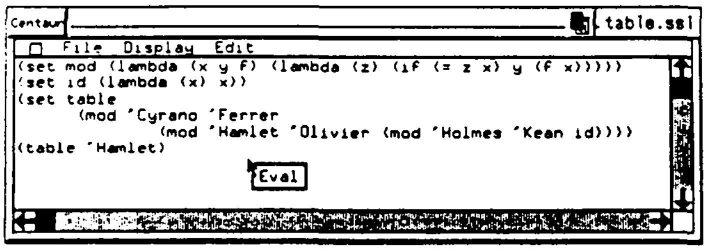
We present the system by showing a brief debugging session. In the first screen (Figure 6), we have started up CENTAUR and opened an editing window. Depressing a mouse button invokes a menu for reading or writing a file.

We have shown only the editing window, omitting the CENTAUR interaction window and other windows that happen to be on the screen. When "**Read**" is selected, we type the name of the file (**table.ssl**) in the CENTAUR interaction window. The "**ssl**" suffix signals CENTAUR to load the SASL parser and pretty-printer, and the "**SASL.env**" file containing LISP code to customize the interface. The file is loaded, parsed, and pretty-printed, leaving the window looking as shown in Figure 7.

This sample program consists of the definitions of the functional **mod**, for function modification at a point, and the identity function **id**. These are used to define a function mapping theatrical characters to the actors who portrayed them. This function, called **table**, is applied to the character's name to get the actor's name. Thus, **(table 'Hamlet)**, the expression appearing in the last line, should evaluate to **Olivier**. However, there is a bug.

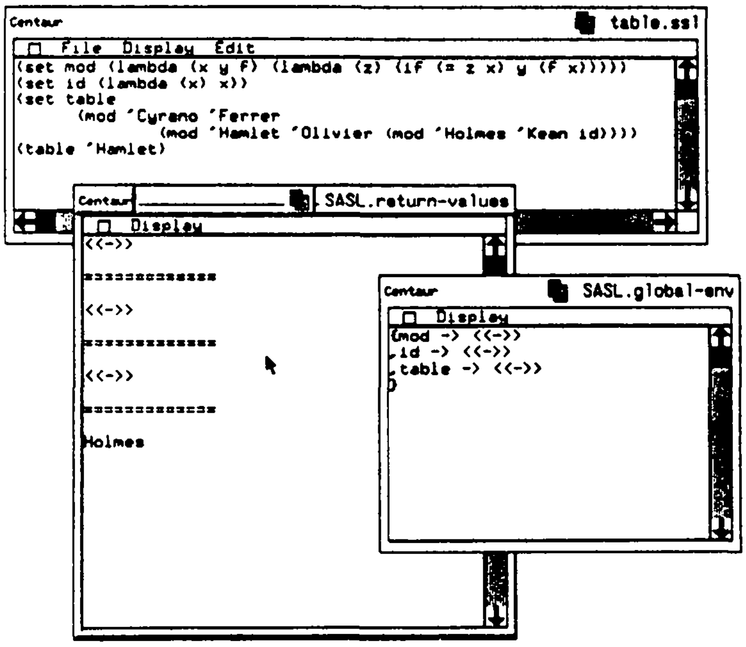
We evaluate the program by depressing the right mouse button which pops up a one-item menu (Figure 8). The TYPOL code is loaded and the program evaluated. Two new windows are created: one to hold the values of the expressions in the program, another to hold the values of global variables (of course, some of the expression values are also the values of global variables). These windows are labeled "**return-values**" and "**global-env**" (Figure 9).

The **return-values** window contains four values separated by double lines. The first three are closures. These are always displayed as "<<->>"; we will see later how to look inside a closure. The fourth value is the atom **Holmes**, which is wrong. The **global-env** window contains bindings for three variables, all bound to closures; these three closures are, of course, the same three closures as given in the **return-values** window.



```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
       (mod 'Cyrano 'Ferrer
             (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

Figure 8: Before running the program

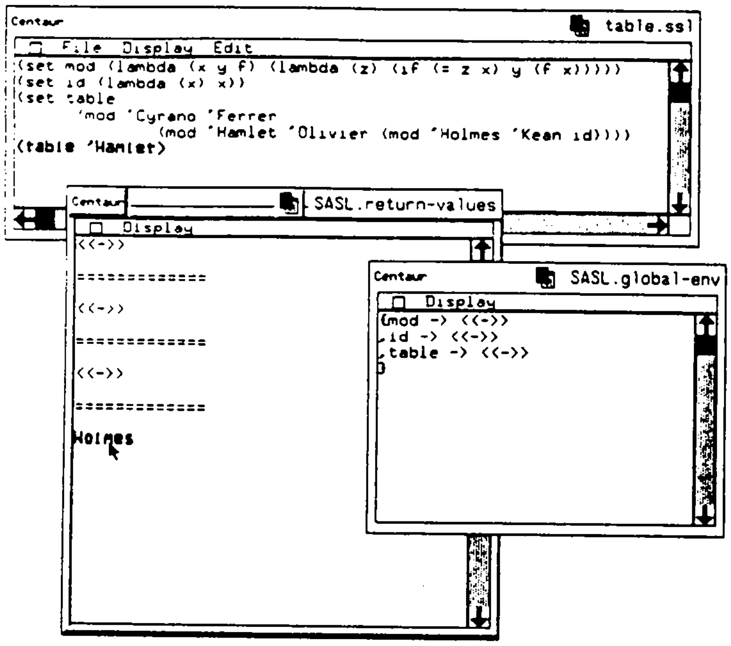


```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
       (mod 'Cyrano 'Ferrer
             (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
<<->>

=============

<<->>

=============

<<->>

=============

Holmes
```

```
(mod -> <<->>
,id -> <<->>
,table -> <<->>
)
```

Figure 9: After running the program



```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
      'mod 'Cyrano 'Ferrer
            (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
<<->>

=============

<<->>

=============

<<->>

=============

Holmes
```

```
{mod -> <<->>
,id -> <<->>
,table -> <<->>
}
```

Figure 10: Highlighting the incorrect value

The purpose of this debugging session, from now on, will be to determine why **(table 'Hamlet)** evaluated to **Holmes**. To put it another way: what is the justification for the value **Holmes**?

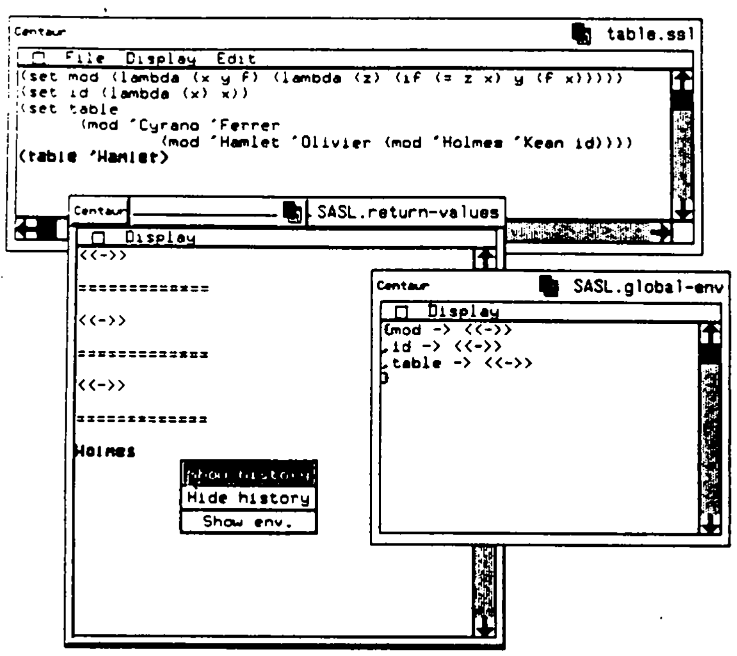
We start our exploration by clicking on the value with the left mouse button. This causes two things to be highlighted: the value itself and the expression, in the **table.ssl** window. The result is shown in Figure 10. Having selected this value, we can request to see its "history" by pulling up a menu with the right mouse button and selecting "**Show history**" (Figure 11).

The display now changes (Figure 12) to show this history, which consists of the series of applications which led to this value. That is, it gives the list of values in the second component of the *XValue* (the subexpression values) and then the history, in the same sense, of the third component (the "control" step), if any. In this case, the value was obtained as the application of a closure to the symbol **Hamlet** (a fact we already know), which led to the evaluation of the body of the closure, which is an application of **if**. We see that the first argument to this **if** was (), or false, so that the second argument is of no interest while the third argument is the value of the application; it is, of course, the (incorrect) value **Holmes**.

The important point to note is that the two applications are applications of values, which can themselves be selected and explored. We start by looking at the closure in the first application. The result of clicking on it with the left button and then pulling up the right button menu is shown in Figure 13. Because the current value is a closure, the menu includes two extra selections: one to see the body of the closure and the other to see its environment. Selecting "**Show closure body**" causes the **lambda** expression stored in the closure to be highlighted (Figure 14). Selecting "**Show closure env**" causes a new window to be popped up in which the stored environment is displayed (Figure 15). In this case, that environment contains bindings for the locally-bound variables **x**, y, and **f**.

We now know exactly what **lambda** expression was applied to **Hamlet** and in what environment. The body of the **lambda** is an **if** application, as shown in the second line of the history of **Holmes**. We now go on to explore that application. The first question to ask is what lead to the condition's having value false ("()"). To explore this, we click on the "()", which causes also the expression "(**= z x**)" to be highlighted. We know that in this expression the **x** has value **Cyrano** (from looking in the **closure-env** window) and **z** has value **Hamlet** (because it was the argument to the closure). However, to see these facts at once, we can raise the popup menu and select "**Show env.**", as shown in Figure 16. (Note that the closure-oriented selections have been removed from the menu because the current value is not a closure.) Another window now appears (Figure 17), giving the environment in which the expression "(**= z x**)" was evaluated to produce value "()".

Up to now, nothing has gone wrong. We expected that the first thing to happen would be the comparison of **Hamlet** to **Cyrano**. Clicking on the small box in the **local-env** window causes it to

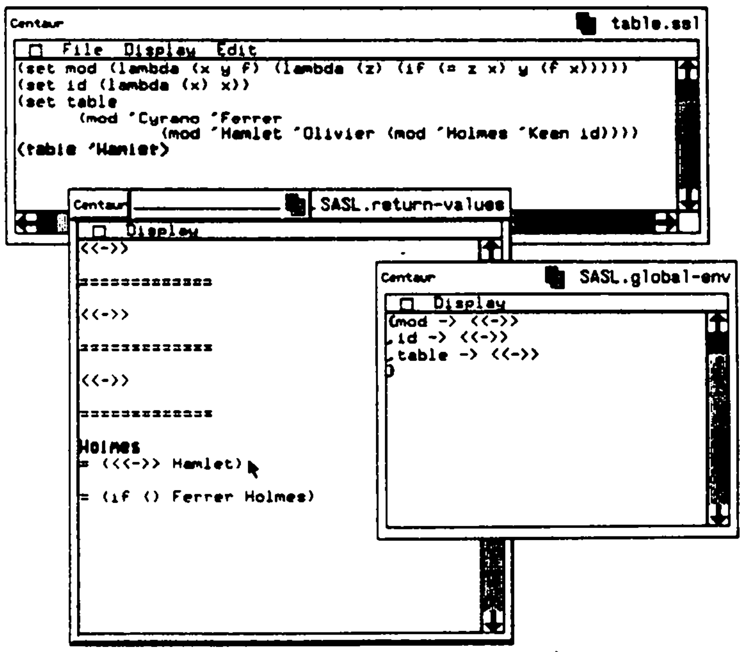


```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
      (mod 'Cyrano 'Ferrer
            (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
<<->>
=============
<<->>
=============
<<->>
=============
Holmes
```

```
(mod -> <<->>
,id -> <<->>
,table -> <<->>
)
```

Figure 11: Trace navigation menu

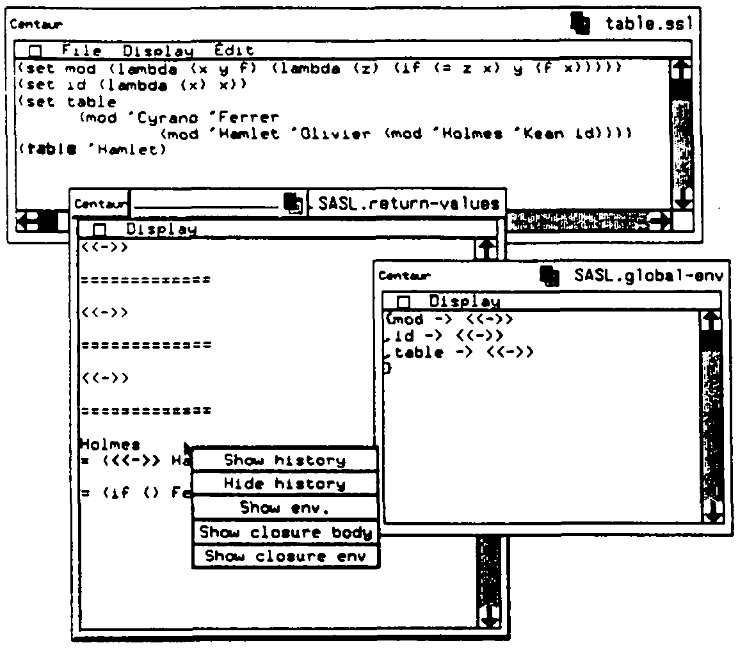


```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
      (mod 'Cyrano 'Ferrer
            (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
<<->>
=============
<<->>
=============
<<->>
=============
Holmes
= (<<->> Hamlet)

= (if () Ferrer Holmes)
```

```
(mod -> <<->>
,id -> <<->>
,table -> <<->>
)
```

Figure 12: Showing the history of **Holmes**

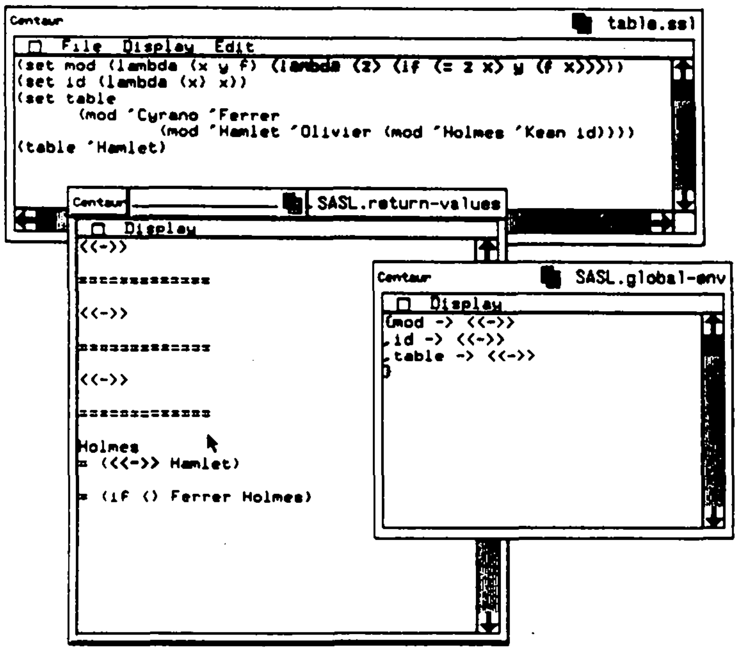


```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
      (mod 'Cyrano 'Ferrer
           (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
<<->>

=============

<<->>

=============

<<->>

=============

Holmes
= (<<->> Ha

= (if () Fe
```

```
{mod -> <<->>
,id -> <<->>
,table -> <<->>
}
```

Figure 13: Exploring a closure

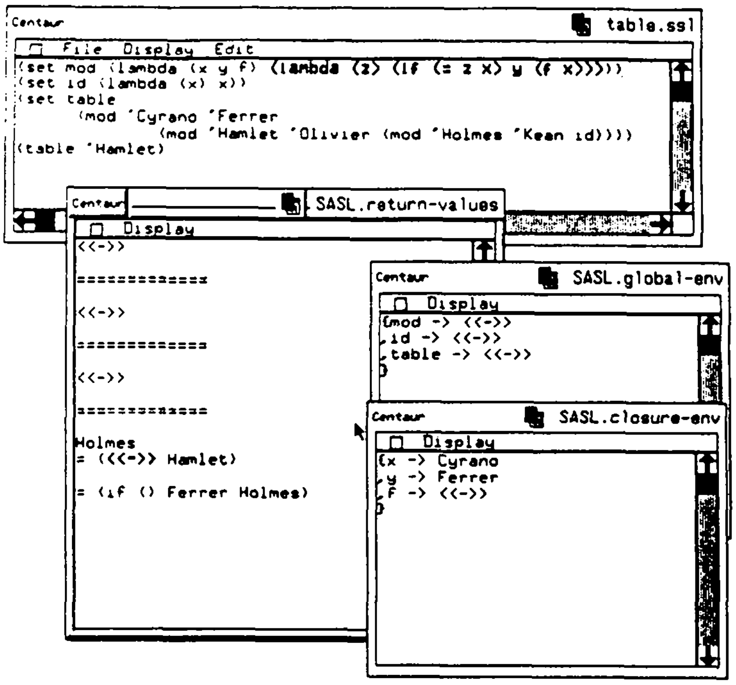


```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
      (mod 'Cyrano 'Ferrer
           (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
<<->>

=============

<<->>

=============

<<->>

=============

Holmes
= (<<->> Hamlet)

= (if () Ferrer Holmes)
```

```
{mod -> <<->>
,id -> <<->>
,table -> <<->>
}
```

Figure 14: Showing the closure's body



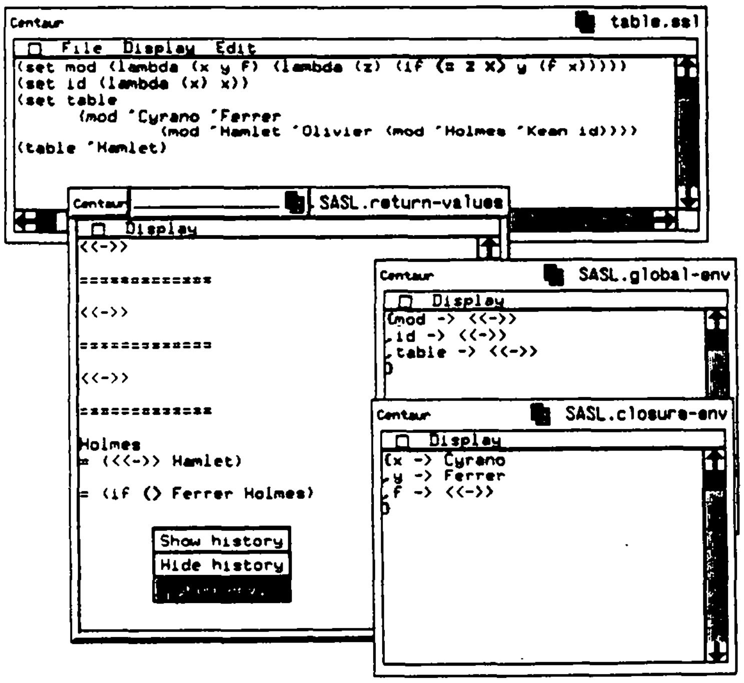
```
Centaur                                   table.ssl
File  Display  Edit
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
     (mod 'Cyrano 'Ferrer
          (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
Centaur                    SASL.return-values
Display
<<->>

=============

<<->>

=============

<<->>

=============

Holmes
= (<<->> Hamlet)

= (if () Ferrer Holmes)
```

```
Centaur                    SASL.global-env
Display
{mod -> <<->>
,id -> <<->>
,table -> <<->>
}
```

```
Centaur                    SASL.closure-env
Display
{x -> Cyrano
,y -> Ferrer
,f -> <<->>
}
```

Figure 15: Showing the closure's environment



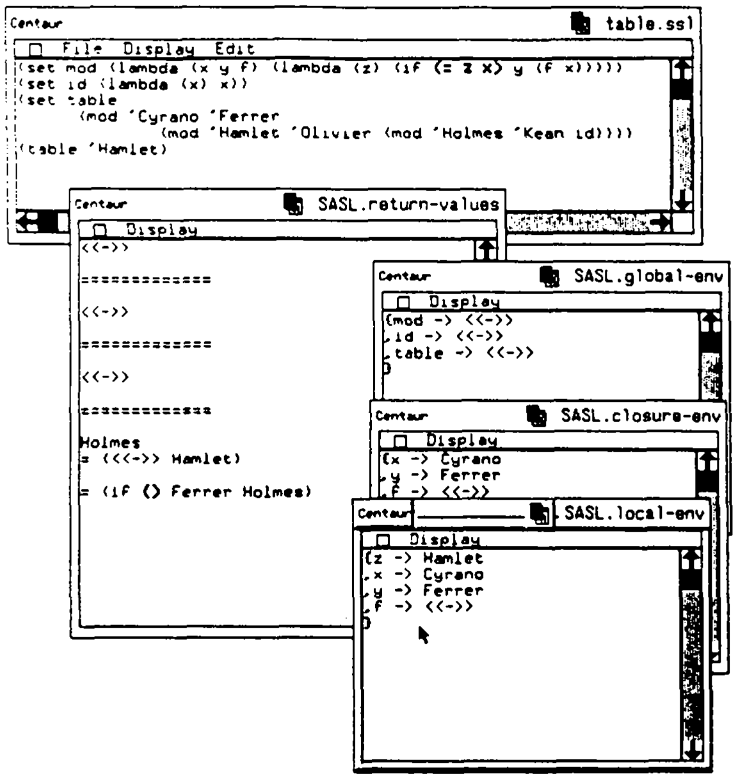
```
Centaur                                   table.ssl
File  Display  Edit
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
     (mod 'Cyrano 'Ferrer
          (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
Centaur                    SASL.return-values
Display
<<->>

=============

<<->>

=============

<<->>

=============

Holmes
= (<<->> Hamlet)

= (if () Ferrer Holmes)

        Show history
        Hide history
        Show env.
```

```
Centaur                    SASL.global-env
Display
{mod -> <<->>
,id -> <<->>
,table -> <<->>
}
```

```
Centaur                    SASL.closure-env
Display
{x -> Cyrano
,y -> Ferrer
,f -> <<->>
}
```

Figure 16: Selecting "Show env."



```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
      (mod 'Cyrano 'Ferrer
            (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
<<->>

=============

<<->>

=============

<<->>

=============

Holmes
= (<<->> Hamlet)

= (if () Ferrer Holmes)
```

```
{mod -> <<->>
,id -> <<->>
,table -> <<->>
}
```

```
{x -> Cyrano
,y -> Ferrer
,f -> <<->>
```

```
{z -> Hamlet
,x -> Cyrano
,y -> Ferrer
,f -> <<->>
}
```

Figure 17: Display of local environment

disappear, and we go on to select the value **Holmes** for exploration (Figure 18). It was obtained as the value of the false branch of the **if**, "**(f x)**", which is now highlighted. Selecting "**Show history**" in the popup menu leads to the display in Figure 19. Again, **Holmes** was obtained by application of a closure (to argument **Cyrano**), whose body was an application of **if**. Looking at the closure by clicking on the "**Show closure body**" and "**Show closure env**" menu selections, we obtain the display shown in Figure 20.

We may look at the closure bound to **f** in the **closure-env** window by clicking on it (Figure 21)[5]

Finally, we realize that the problem is in the closure's argument **Cyrano** (Figure 22). It should have been **Hamlet**, which is to say, the **x** in **(f x)** should have been **z**.

6 Conclusions

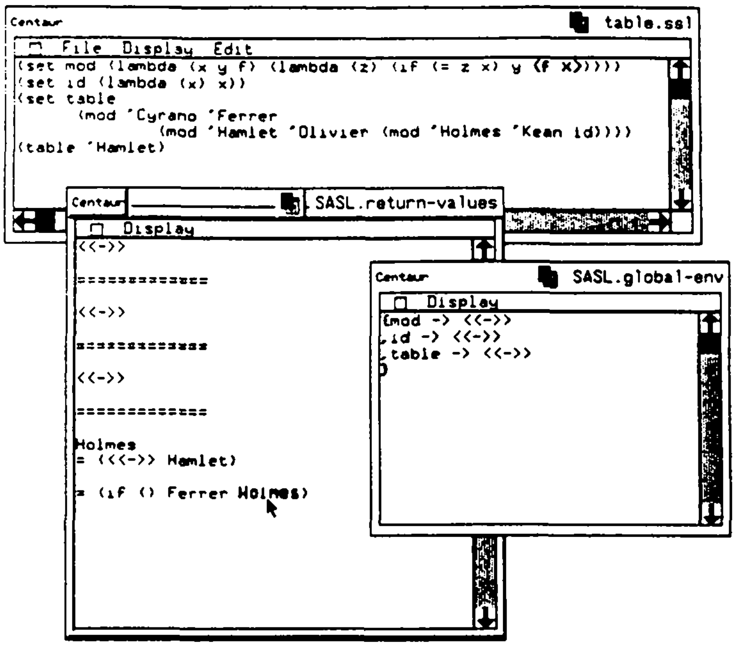
We have presented a trace-based debugging environment for a simple lazy, functional language. We argued that traces are a natural, even inevitable, approach to debugging of lazy languages, because stop-and-examine techniques run up against the unpredictability of lazy evaluation. Our definition of "trace" was given and our system demonstrated on a small program.

Our prototype was built using the CENTAUR system. This allowed us to defined the trace semantics of the language using natural semantics and provided various window creation and customization primitives. Most importantly, it gave us PPML for displaying the traces, which handled customization of the display as well as pointing. Aside from the METAL, PPML, and TYPOL specifications, we needed about 550 lines of LISP to complete our system.

The overall goal of this work has been to demonstrate a "hypertextual" approach to trace-based debugging. Our argument is that this overcomes one of the most serious problems traditionally associated with traces: information overload.

It is too early to call this experiment a success, or to say with certainty that trace-based debugging promises to be a practical alternative for debugging lazy languages in the future. Our system is "highly experimental," which is computer science code for inefficient and buggy. There are three serious problems with the current implementation:

[5]Note that the closure in the **return-values** window remains highlighted; this is a bug, not a feature.

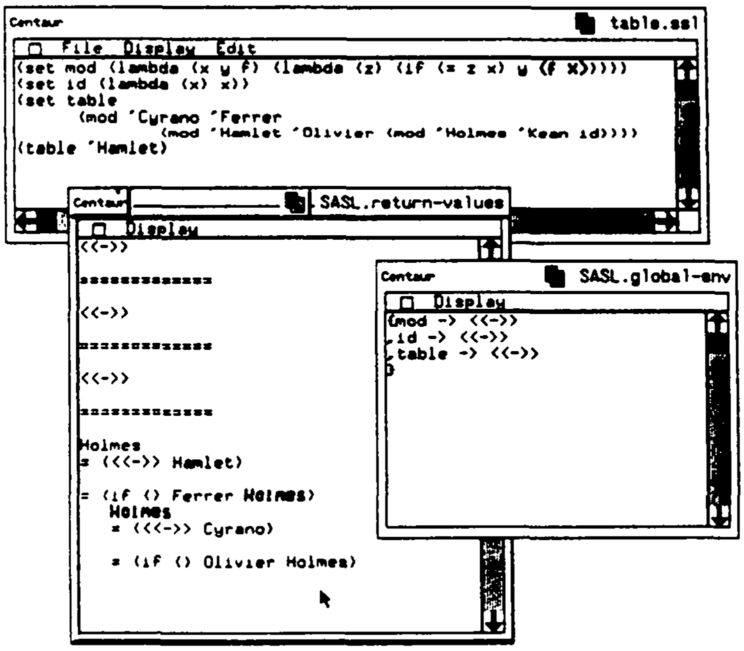


```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
      (mod 'Cyrano 'Ferrer
           (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
<<->>

=============

<<->>

=============

<<->>

=============

Holmes
= (<<->> Hamlet)

= (if () Ferrer Holmes)
```

```
(mod -> <<->>
,id -> <<->>
,table -> <<->>
)
```

Figure 18: Exploring value Holmes

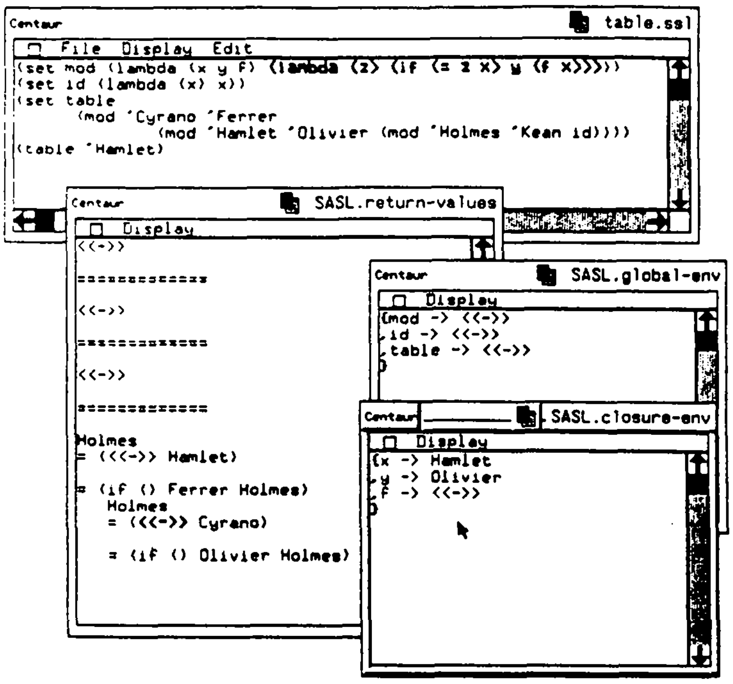


```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
      (mod 'Cyrano 'Ferrer
           (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
<<->>

=============

<<->>

=============

<<->>

=============

Holmes
= (<<->> Hamlet)

= (if () Ferrer Holmes)
    Holmes
    = (<<->> Cyrano)

    = (if () Olivier Holmes)
```

```
(mod -> <<->>
,id -> <<->>
,table -> <<->>
)
```

Figure 19: Showing history recursively



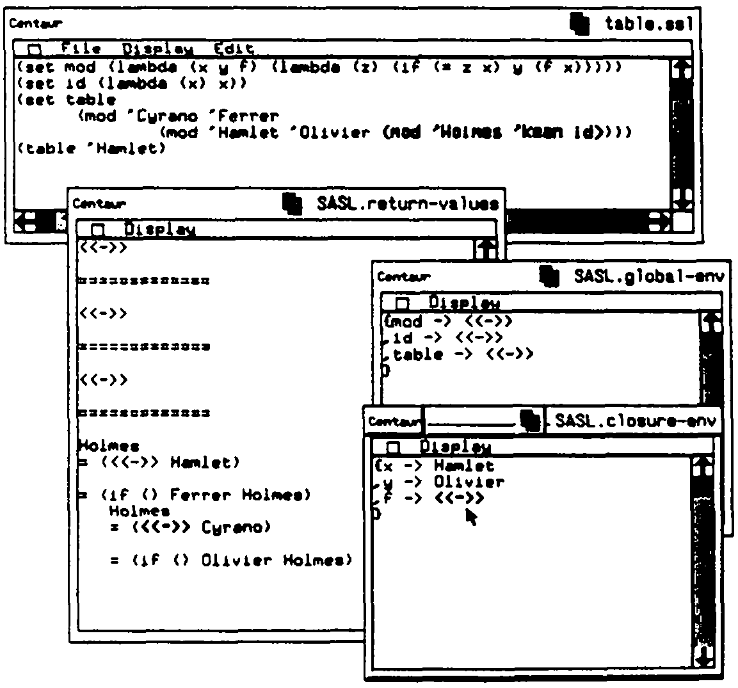
```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
      (mod 'Cyrano 'Ferrer
            (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
<<->>
============
<<->>
============
<<->>
============
Holmes
= (<<->> Hamlet)
= (if () Ferrer Holmes)
  Holmes
  = (<<->> Cyrano)
  = (if () Olivier Holmes)
```

```
(mod -> <<->>
,id -> <<->>
,table -> <<->>
)
```

```
(x -> Hamlet
,y -> Olivier
,f -> <<->>
)
```

Figure 20: Exploring a closure



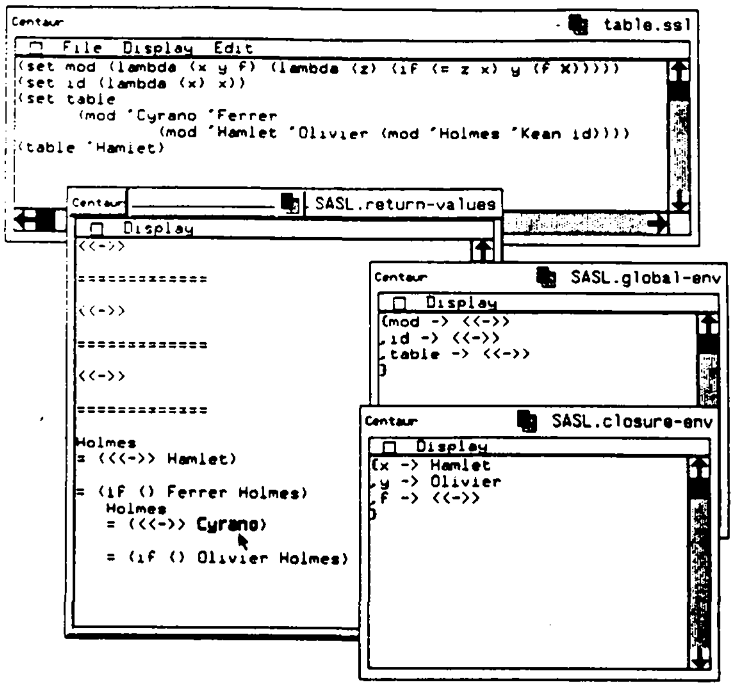
```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f x)))))
(set id (lambda (x) x))
(set table
      (mod 'Cyrano 'Ferrer
            (mod 'Hamlet 'Olivier (mod 'Holmes 'kean id))))
(table 'Hamlet)
```

```
<<->>
============
<<->>
============
<<->>
============
Holmes
= (<<->> Hamlet)
= (if () Ferrer Holmes)
  Holmes
  = (<<->> Cyrano)
  = (if () Olivier Holmes)
```

```
(mod -> <<->>
,id -> <<->>
,table -> <<->>
)
```

```
(x -> Hamlet
,y -> Olivier
,f -> <<->>
)
```

Figure 21: Exploring a value in another window



```
(set mod (lambda (x y f) (lambda (z) (if (= z x) y (f X)))))
(set id (lambda (x) x))
(set table
      (mod 'Cyrano 'Ferrer
            (mod 'Hamlet 'Olivier (mod 'Holmes 'Kean id))))
(table 'Hamlet)
```

```
<<->>

=============

<<->>

=============

<<->>

=============

Holmes
= (<<->> Hamlet)

= (if () Ferrer Holmes)
   Holmes
   = (<<->> Cyrano)

   = (if () Olivier Holmes)
```

```
(mod -> <<->>
,id -> <<->>
,table -> <<->>
)
```

```
(x -> Hamlet
,y -> Olivier
,f -> <<->>
)
```

Figure 22: Argument to closure is wrong

- Memory usage is so high that only very small problems can be run. In part, this is to be expected, since the memory is going to be proportional to the number of computation steps. However, the current implementation greatly exaggerates the memory needs, for this reason: Despite the apparently tree-like nature of traces, they are not in fact trees but DAG's. For example, a single environment can be included in several closures and thunks; moreover, every reference to a variable results in a reference to its *XValue* being stored in the trace. However, AST's are, by definition, trees. The evaluation of a SASL expression actually is done in PROLOG, and this computation creates the trace as a DAG, but the **sendtree** "flattens" it into a VTP tree, resulting in an enormous combinatorial expansion. What is needed is support for DAG's in the VTP, but CENTAUR is unlikely to evolve in that direction in the foreseeable future. Alternatively, the TYPOL trace semantics could be modified to avoid sharing, as indeed has already been done to some extent (for example, the abstract syntax operator **envintro** was introduced for this purpose); it is not clear how difficult this would be or how badly it would affect the readability of the semantics.

- The current definition of trace, and the procedures for trace navigation in the system, are not terribly natural. A good deal of further experimentation is needed to find a natural level of abstraction for user interactions.

- The system cannot be used to debug programs that loop. When this happens, the program never produces any trace, so there is nothing to look at. This could be fixed by creating the trace incrementally — though this is technically very tricky — and allowing the user to interrupt a program's execution. The PPML specifications would also have to be modified to account for partial traces.

We have given no consideration to how traces might be computed in a real system. No implementation scheme will be able to avoid an exorbitant use of memory, but we would calculate that this might average about four words per execution step, which would allow for quite a few steps considering memory sizes of modern computers. A method to control the generation of traces could presumably reduce this figure significantly.

In conclusion, the hypertext approach to trace-based debugging seems promising. This is a rather unusual hypertext application, in several ways: the "document" is computer-generated rather than human-generated; it is a "small node" document in contrast to most hypertext documents, in which a node consists of an entire paragraph or screenful of data; it is a DAG, not a tree nor even a tree with occasional cross-references. There does not appear to be any existing hypertext system that is designed to handle such documents.

Acknowledgements

This work was carried out while the author was on sabbatical from the University of Illinois, working in the CROAP group at INRIA, which developed CENTAUR. He would like to acknowledge the support provided by Gilles Kahn, the leader of CROAP, as well as the technical assistance of Gilles, Dominique Clément, Yves Bertot, Laurence Rideau, Janet Incerpi, Francis Montagnac, Laurent Hascoët, Laurent Théry, Joelle and Thierry Despeyroux, and Didier Austry.

References

[1] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, "Centaur: the system," Proc. ACM Software Eng. Symp. on Practical Software Development Environments (SIGSOFT '88), Boston, Nov. 1988, 14-24.

[2] CENTAUR documentation (User's Guide, User's Manual, and Reference Manual), Version 0.9, June 1989 (*inquire at* `centaur@mirsa.inria.fr`).

[3] T. Despeyroux, "Executable specifications of static semantics," in **Semantics of Data Types**, Lecture Notes in Computer Science 173, Springer-Verlag, June 1984.

[4] P. Hudak, P. Wadler (eds.), **Report on the Programming Language Haskell, Version 1.0**, April 1990.

[5] G. Kahn, "Natural Semantics," in K. Fuchi, M. Nivat (eds.), **Programming of Future Generation Computers**, Elsevier, 1988, 237-258.

[6] G. Kahn, B. Lang, B. Mélèse, "METAL: a formalism to specify formalisms," Science of Computer Programming 3, 1983, 151-188.

[7] S. Kamin, **Programming Languages: An Interpreter-based Approach**, Addison-Wesley, Reading, Mass., 1990.

[8] B. Lang, "The Virtual Tree Processor," in J. Heering, J. Sidi, A. Verhoog (eds.), **Generation of Interactive Programming Environments**, Intermediate report, CWI Report CS-R8620, Amsterdam, May 1986.

[9] J.T. O'Donnell, C.V. Hall, "Debugging in applicative languages," Lisp and Symbolic Comp. 1, 1988, 113-145.

[10] G.D. Plotkin, "A structural approach to operational semantics," DAIMI FN-19, Computer Science Department, Aarhus Univ., Aarhus, Denmark, Sept. 1981.

[11] I. Toyn, C. Runciman, "Adapting combinator and SECD machines to display snapshots of functional computations," New Generation Comp. 4, 1986, 339-363.

[12] D. A. Turner, *SASL Language Manual*, Computer Laboratory, U. of Kent, Canterbury, England, originally dated 1976, revised 1979 and 1983.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique .

ISSN 0249-6399