Working Paper Series
ISSN 1170-487X

MiraCalc: The Miranda Calculator The Unix Version

by Doug Goldson, Mike Hopkins & Steve Reeves

Working Paper 94/5
April, 1994

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

MiraCalc: The Miranda [1] Calculator
The Unix Version

Doug Goldson
Key Centre of Design Computing,
Department of Architecture and Design Science,
University of Sydney, Australia
douglas@archsci.arch.su.edu.au

Mike Hopkins
Department of Computer Science,
QMW, University of London, U.K.
leonardo@dcs.qmw.ac.uk

Steve Reeves
Department of Computer Science,
University of Waikato,
Hamilton, New Zealand
stever@waikato.ac.nz

April 1994

1. What is MiraCalc?

Those of you who already have some experience of programming, or experience of simply using a computer, will know that computers can be very unforgiving. They are fussy, and unless you get things exactly right they will complain. The program described in this document has grown out of an attempt to help you to understand what is going on when the computer complains. It is designed to help you with the Miranda scripts that you will be writing as part of the process of learning Miranda

We have called our program "*MiraCalc*" (Miranda Calculator) since we hope that you will use it in the same way as you might use an arithmetic or scientific calculator, except that instead of calculating with numbers, our calculator works with Miranda expressions. We hope that you will experiment with *MiraCalc* and use it to explore and learn about Miranda.

The calculations of which *MiraCalc* is capable are of three sorts:

to calculate the type of Miranda expressions

to calculate the scope of names in Miranda expressions

to calculate the value of Miranda expressions

---

[1] Miranda is a trademark of Research Software Ltd.

So, one function of *MiraCalc* is to help you to understand the role that *type* plays in Miranda. In arithmetic there are only three types of object, *rational numbers*, *real numbers* and *number functions* (operations on numbers) but in Miranda there are lots of different kinds or types of object including, of course, the *numbers* and *functions* of arithmetic. Introducing lots of new objects means that we can do lots more calculations than those of arithmetic but it also raises a danger, since it makes it more likely that we will mix these objects up in cocktails of nonsense. In order to avoid, or rather to detect, these nonsense combinations Miranda has a discipline of types.

The second function of *MiraCalc* is to allow you to investigate the notion of scope. A name is in scope if it can be given a value by the system. Some names are always in scope because their definitions are supplied by the Miranda system. They are part of the *standard Miranda environment* or, simply, the *standard environment*. The system always works in a *current environment* which is at least as large as the standard one, in the sense that it contains all the objects defined in the standard environment and perhaps more besides. The effect of successfully compiling a script is to extend the standard environment with the new objects which are defined in the script. *MiraCalc* allows you to ask if names in your script occur free, bound or binding.

The third function of *MiraCalc* concerns the values of expressions. Just as a pocket calculator works out that 2^{3^2} equals `512`, we would like our calculator to work out that the value of `2^3^2` is `512`. But since the *Miranda system* itself already provides a means of calculating the value of Miranda expressions, why have we bothered to write our own calculator? The reason is because we believe it is often very useful to watch a calculation proceed one step at a time whereas the calculation which the Miranda system performs is always silent, giving the appearance that it occurs 'all at once'.

In conclusion, *MiraCalc* is intended to help you to investigate the scope, meaning and value of Miranda expressions in the same way as an ordinary pocket calculator allows you to investigate the structure and value of arithmetic expressions.

2. Using *MiraCalc*

MiraCalc has seven menus - `File`, `Edit`, `Font Size`, `Navigate`, `Utilities`, `Calculate` and `Help`.

Since *MiraCalc* uses different sorts of window, for example a script window or a calculation trace window, and since each sort of window can only meaningfully have certain operations applied to its contents, the options available to you from the menus and buttons will change as the choice of the currently selected window changes. For example, if the script window is selected then, as far as the `Calculate` option is concerned, at first you can only parse it[2]; not until that has been successfully completed can you do an evaluation since that option does not make sense when applied to an unparsed script.

An existing file can be edited using `Open` in the `File` menu to open the file and display its content in a window and a new file can be created using `New`. Files are closed and saved using `Close...`, `Save` and `Save as...` . When you are ready to finish a session, use `Exit` in the

[2] In fact you can choose `Evaluate` at this point, but you will only be able to refer to names defined in the standard environment.

`File` menu. You will be asked, for each open window that has changed during the session, whether or not you want to save the changes.

The `Edit` menu offers the standard editing options. Thus it is possible to `Cut`, `Copy` and `Paste` text in a window and to search and replace selected text. Since *MiraCalc* is a program editor it is also convenient, in addition to these options, to have a `Line number` option which moves the cursor to a selected line in a window. This is useful because line numbers are reported in diagnostic messages where there are errors in the script.

There is also a `Font Size` option, which allows you to change the size of the font, which you might want to make bigger so that the script is more easily readable, or smaller so that you can see tmore of a script by having more of it visible at once.

The `Navigate` menu allows you to move to the top or bottom of a file, to move to where the cursor is currently situated and to move to a line number of your choice.

The `Utilities` menu allows you to find and change text in the script window.

The `Help` menu offers brief reminders of the function of menu options.

The `Calculate` menu allows you to explore the structure and meaning of Miranda scripts. The menu options are explained in the next section.

3. Sorts of calculation

3.1 Parse

The `Parse` option in the `Calculate` menu allows you to parse a Miranda script. Parsing is the activity of reading a collection of symbols according to a particular syntax of expressions. The `Parse` option reads the current script window as if it were a Miranda script, i.e. according to the syntax rules of Miranda. If *MiraCalc* succeeds in this then your script is syntactically correct or *well-formed*. For a script to be meaningful it must be well-formed. If your script is not well-formed then a message will appear in the lower window that says there is an error in your script together with some diagnostic information.

For example, after parsing the script

```
badsyntax = 1 + @
```

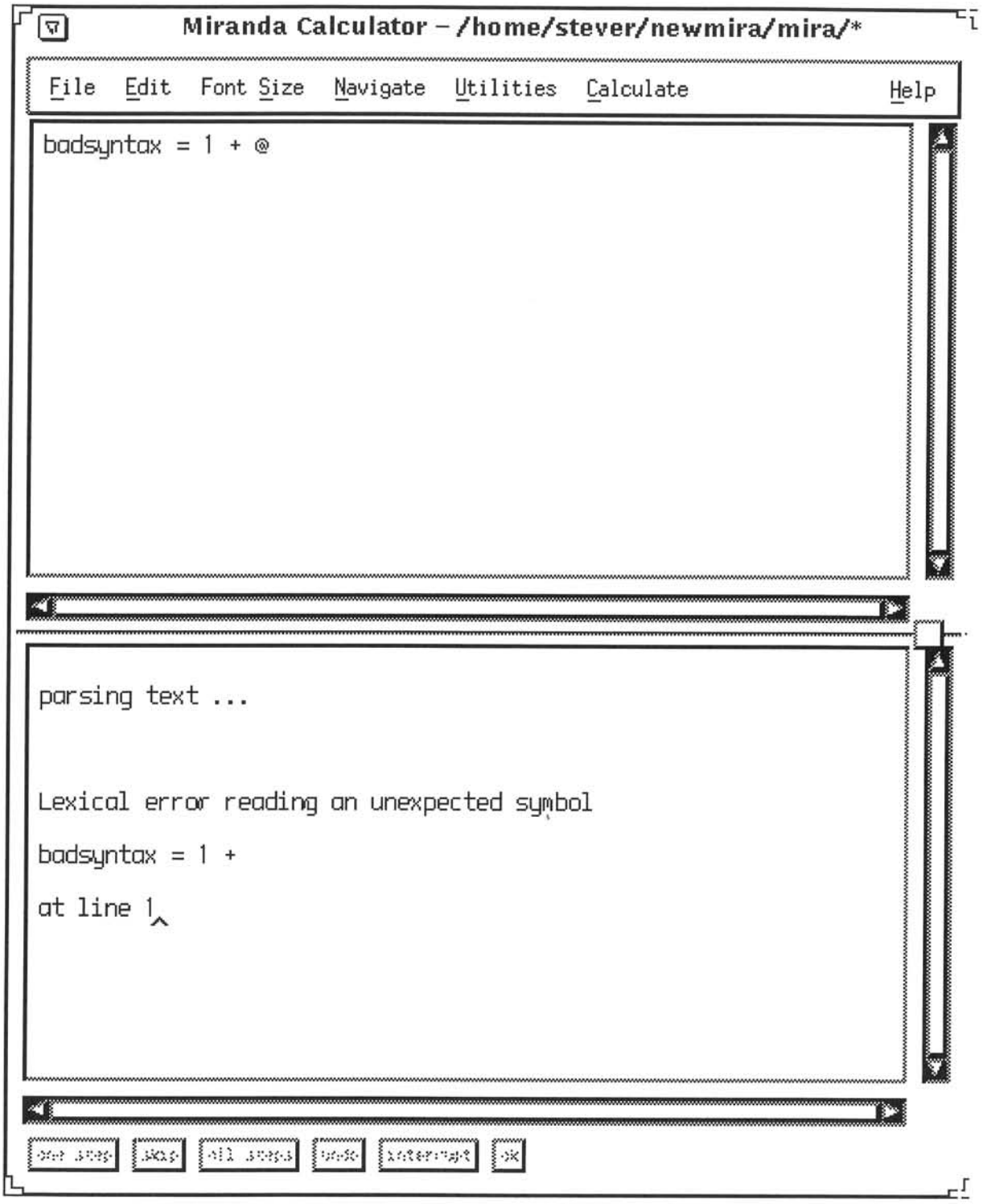
your screen should look like the picture on the next page.

Not every well-formed script is meaningful. Try parsing the following script

```
badtype = 1 + 'a'
```

As well as being well-formed a script must also be *well-typed*. If your script is then it is type-checked to ensure that it is well-typed too. If type-checking is successful then you can be sure that your script is executable. Again, if the script is not well-typed then a message will appear in the lower window.

If the content of your script window is both well-formed and well-typed then it is possible to use the other items of the `Calculate` menu to explore its structure.

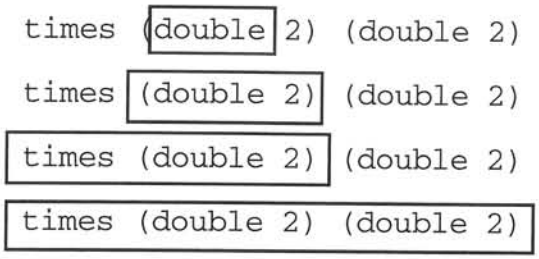


3.2 Expand and Contract

The `Expand` and `Contract` options operate on the parsed version of your source. They are meant to illustrate the structure of program expressions, the grammar of definitions, the associativity and precedence of operators, and the structure of curried functions and functions defined using patterns. `Expand` is used to expand expressions, `Contract Left` and `Contract Right` to contract expressions. `Back` can be used to go back to previous states of the cursor.

`Expand` works as follows. It takes the current cursor selection in the script window and then expands this selection to cover the smallest enclosing expression. For example, using a box to

represent the extent of highlighted text in a window, the effect of successive expansions can be illustrated as



This example illustrates the fact that application associates to the left.

Some operations, like `*`, are written infix. Others, like application, are written prefix. However, there is nothing essentially infix about `*` and it is just a matter of convention that we write it `1 * 2` instead of `* 1 2`. The same applies to the other arithmetic operators, allowing

```
2 * x^2 + 3 * y
```

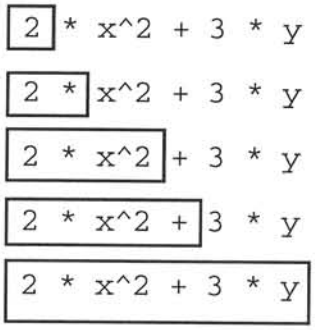
to be written

```
+ * 2 ^ x 2 * 3 y
```

i.e.

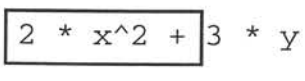
```
+ (* (^ x 2)) (* 3 y)
```

MiraCalc's internal representation of expressions is prefix which explains the following sequence of expansions



As you would expect, the `Contract` options of the `Calculate` menu offer the inverse of `Expand`, with the choice of contracting to the left-hand sub-expression or the right-hand one (assuming they exist).

For example, given the last expansion in the expression above, `Contract Left` gives



whereas `Contract Right` gives

Lists are built-in Miranda data structures which are built up from the empty list, `[]`, and the list constructor `:`, pronounced "cons". So, if `l` is a list then `x : l` is a list too, provided that `x` has the same type as the type of all the list elements of `l`. Miranda has two ways of representing lists, one uses `:` and `[]`, for example

```
'M' : 'i' : 'r' : 'a' : 'n' : 'd' : 'a' : []
```

the other uses a shorthand

```
['M', 'i', 'r', 'a', 'n', 'd', 'a']
```

It is important to realise that the shorthand is just another way of writing the *same* list, i.e.

```
['M', 'i', 'r', 'a', 'n', 'd', 'a'] =
'M' : 'i' : 'r' : 'a' : 'n' : 'd' : 'a' : []
```

Tuples are also built-in Miranda data structures. Unlike lists, the individual elements of tuples may have different types, for example

```
(['M','i','r','a','n','d','a'],(remove,"Miranda"),+)
```

This is a triple, or 3-tuple. The first element is a list of characters, the third element is a numeric function and the second element is itself a tuple, a 2-tuple whose first element is a function and whose second is a list of characters.

Tuples raise a problem for `Expand` and `Contract` because, unlike lists, they are not defined as binary structures. Tuples are n-ary structures where n = 0 or n ≥ 2. Defining the left and right contraction of a list `x : l` is simple. It is `x :` and `l` respectively (remembering that Miranda operators are curried) but how do we define the left and right contraction of `(x,y,z)`? The answer is that we cannot, so we avoid the problem by making the contraction of `(x,y,z)` itself. For similar reasons the expansion of, say

```
(x , y , z )
```

is

```
(x , y , z)
```

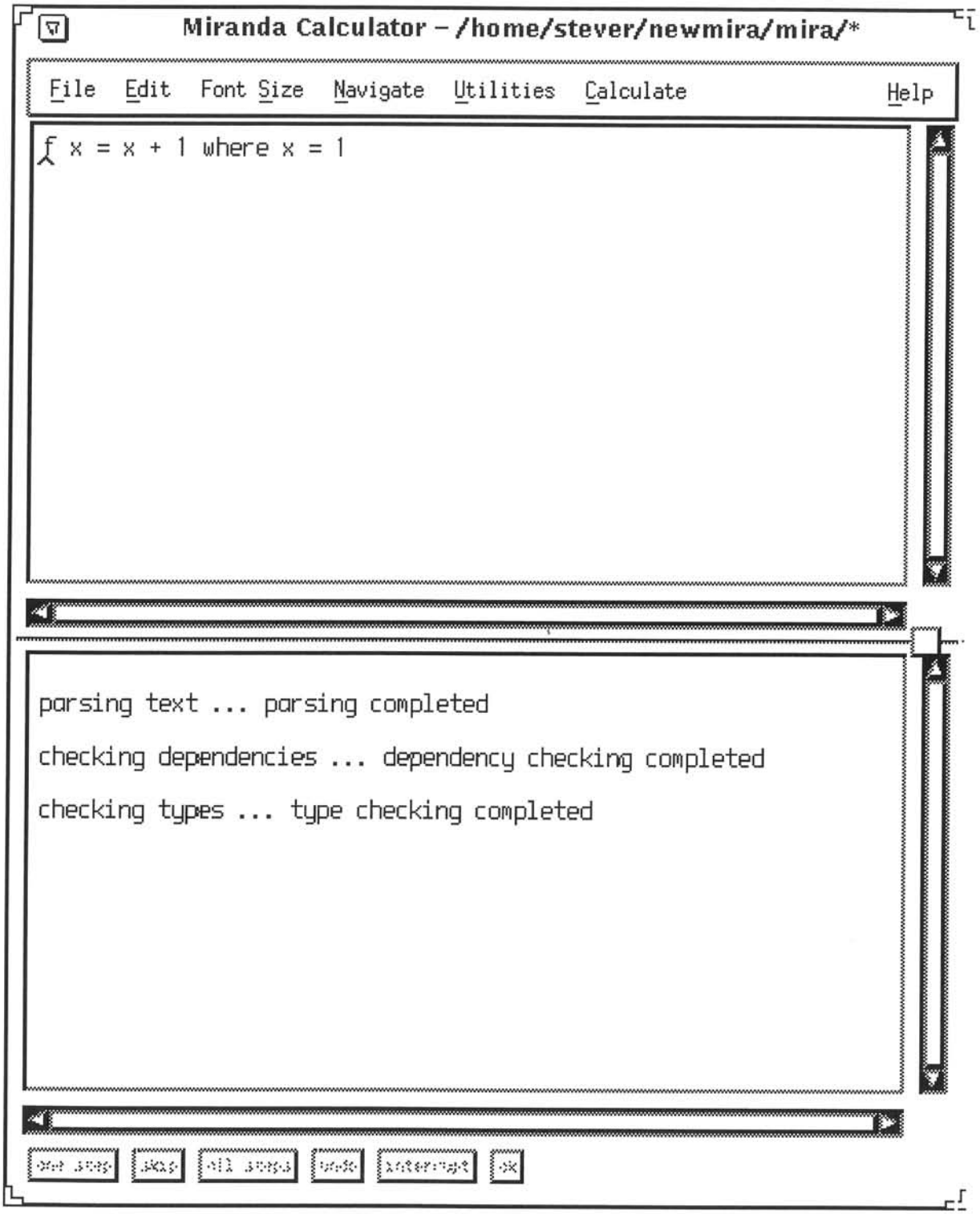
3.3 Scope: Free, Bound and Binding

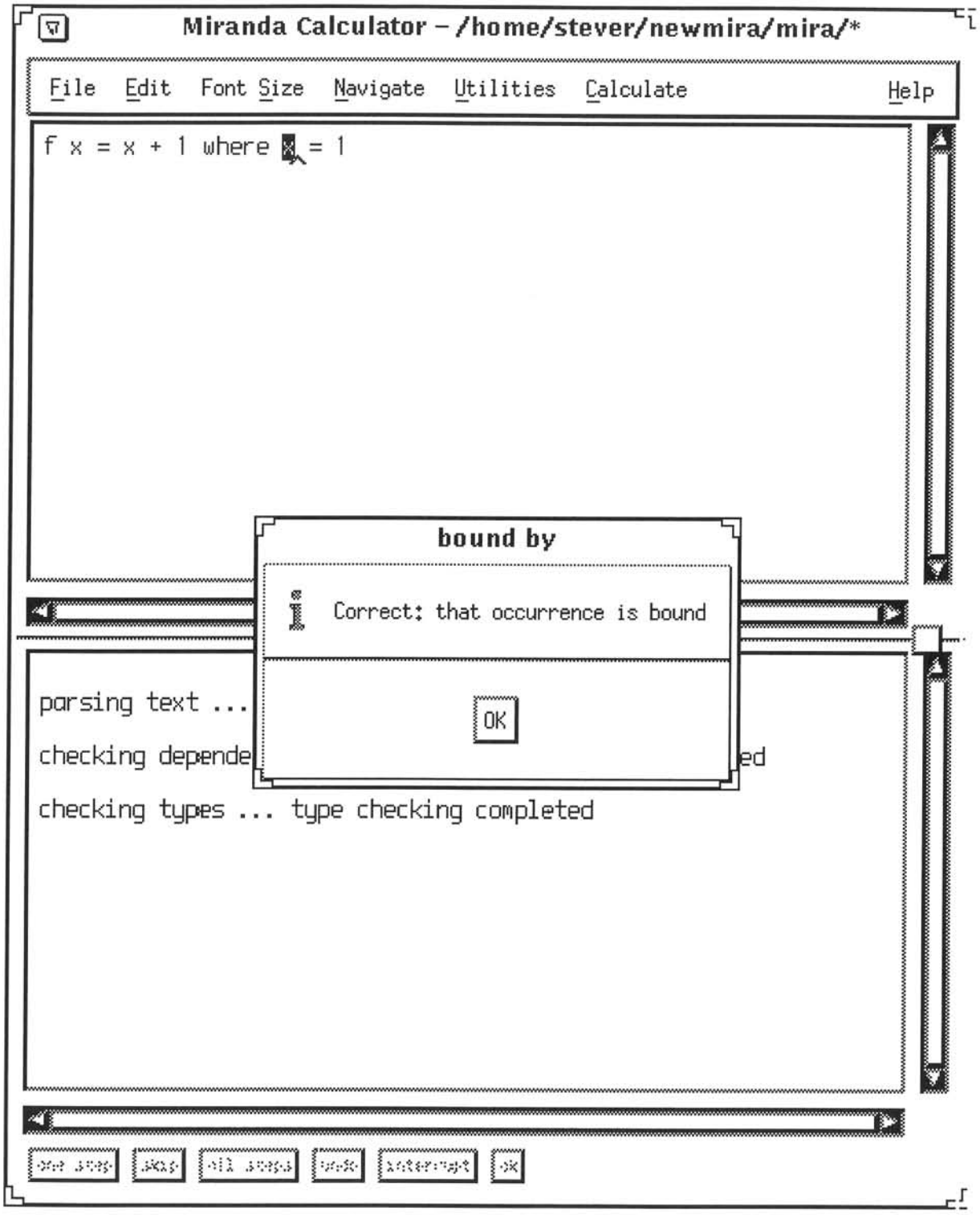
As you learn more about Miranda you will learn how to determine when an occurrence of a name is free, bound or binding in an environment and you will see that it is sometimes quite hard to calculate. For example, which binding occurrence of `x` binds the second occurrence of `x` in

```
f x = x + 1 where x = 1
```

and does `f 2` have the value `2` or `3`? To answer this question we can experiment with *MiraCalc*. First you need to open a new file using `New...` from the `File` menu. Call this file `example1`. Now type in the definition of `f` given above and parse it using the `Parse` option from the `Calculate` menu. Your screen should look like the first picture below. Now select the second occurrence of `x` and check to see that it is bound by selecting the `Bound` option from the `Calculate` menu. A message is displayed confirming your guess as correct and,

importantly for deciding the binding of this x, the cursor selection in the window is moved to cover the *binding* occurrence of x. The window will now look like the second picture.





3.4 Type

MiraCalc allows you to check your knowledge of types using the `Show Type` or `Guess Type` options in the `Calculate` menu. In order to determine the type of an expression you first select it using the cursor and then choose the `Show Type` option. A dialogue appears telling you the type of the expression.

Using the `Guess Type` option creates a dialogue which invites you to guess the type of the selected expression. If your guess is correct a message appears to confirm it. Otherwise, a diagnostic message is given which includes the correct type of the selected expression. This

option is sometimes more helpful than `Show Type` since it gives limited information about why a guess is wrong (if it is). It may be useful to use `Expand` and `Contract` to make the initial selection.

3.5 Evaluate

You can choose this option even if you have not yet parsed a script because the standard Miranda environment is always available as a calculational environment and so, as long as you only refer to expressions defined there, you can evaluate them in the absence of a parsed script.

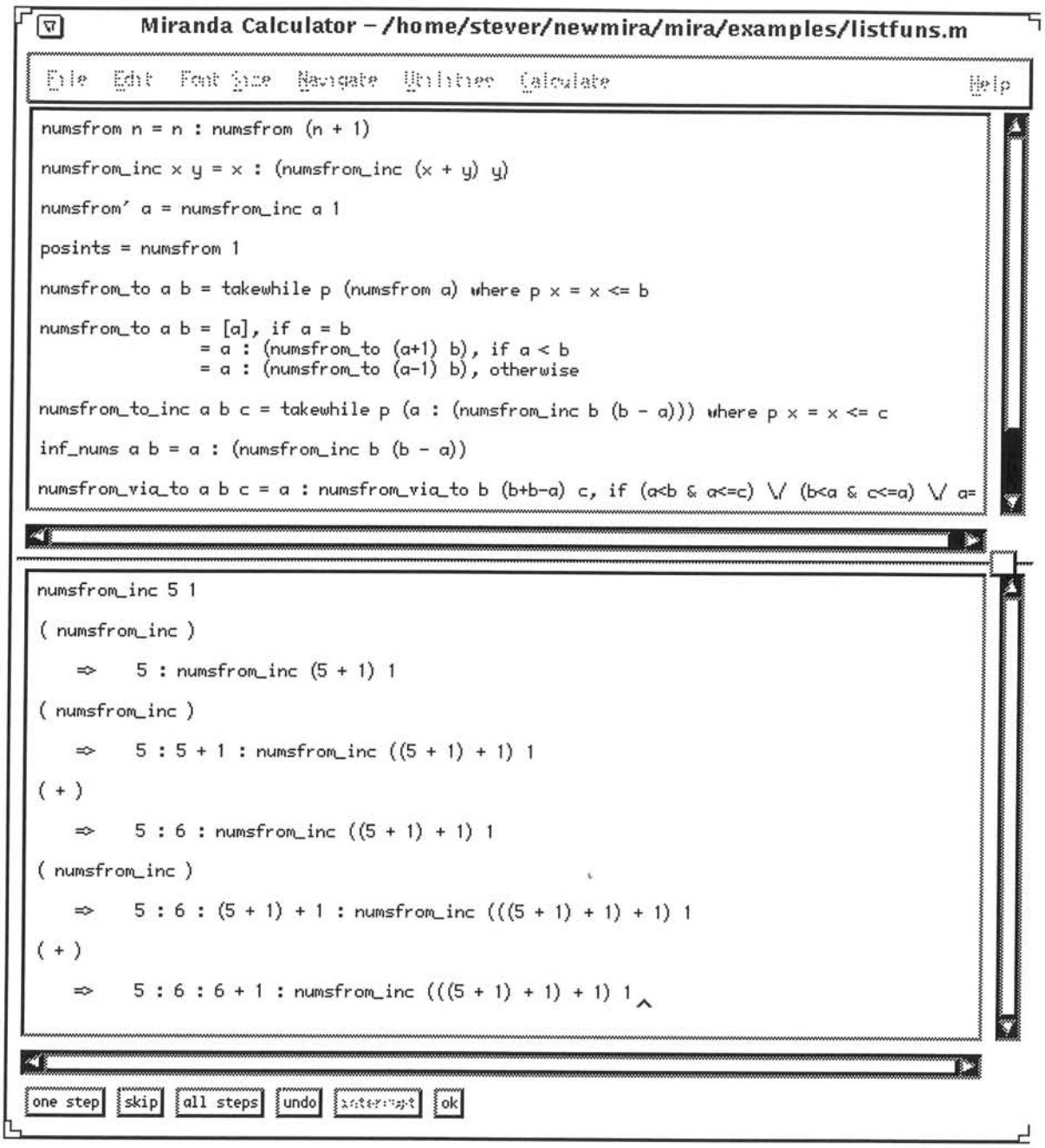
However, to show how this option can be used we will assume the existence of a file called '`listfuns.m`' which has been opened and parsed. (You can examine the script by looking at the picture below.) If you choose the `Evaluate` option you will get a dialogue asking for an expression to evaluate. Let us assume that you ask for the expression

```
numsfrom_inc 5 1
```

to be evaluated, then your screen will look like

The `One Step` button causes *MiraCalc* to do a single step in the calculation, and the result is displayed in the lower window as shown in the first picture below. Further choices of `One Step` lead to a calculation window which looks like the second picture.

The `Skip` button causes *MiraCalc* to do a 'larger' step in the calculation of a value. In fact, sometimes its effect will be the same as `One Step` but in general it will have a larger effect. It can be useful in skipping the 'uninteresting' aspects of a calculation in order to focus on its 'interesting' parts. The precise behaviour of `Skip` is best determined by experimentation.



The `All Steps` button performs calculation steps silently, like Miranda, until an expression is fully evaluated. Since some expressions, like `numsfrom 5`, are never fully evaluated, it is possible that an all-steps calculation may never terminate so that if you leave the calculation for long enough you will eventually get a message telling you that *MiraCalc*'s memory is exhausted. If you get bored waiting for an all-steps calculation to terminate you can press the `interrupt` button to stop it.

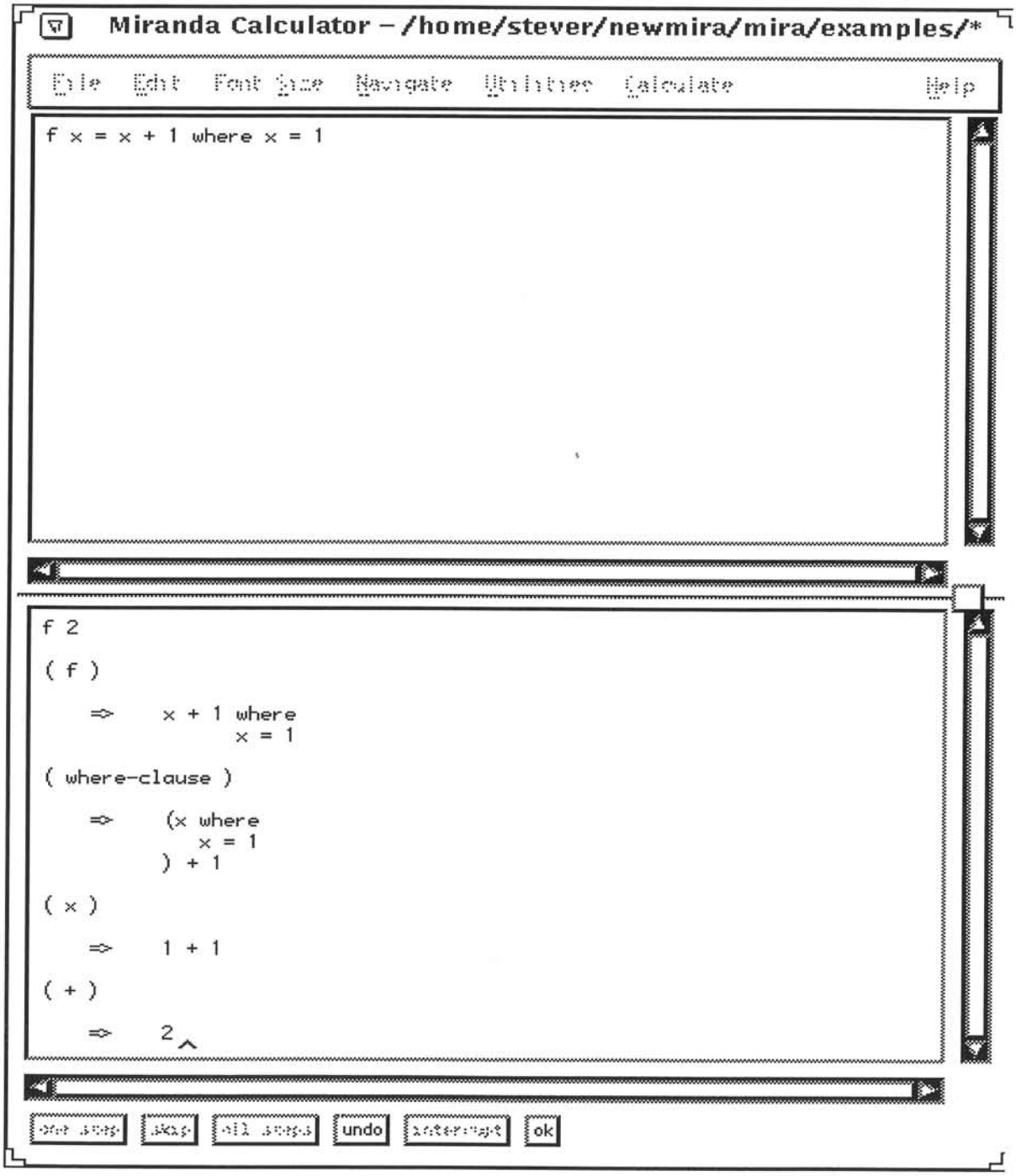
The `Undo` button deletes the current step of the calculation and takes you back to the previous one.

4. New expressions

In order to be able to display step-by-step calculations in all circumstances it was necessary to invent some new notation which extends the language of Miranda *expressions*.

4.1 Embedded `where`-expressions and renaming

Returning to the example of §3.3 we can use the `Evaluate` option to calculate the value of `f 2`. The window looks like



The second step in this calculation shows how a `where`-expression is evaluated. A `where`-expression is an expression of the form

```
expression where defs
```

For example

```
x + 1 where x = 1
```

MiraCalc uses two rules to evaluate `where`-expressions and which rule applies depends upon the form of the given expression. The rule applied to this expression moves the `where`-clause `where x = 1` into the body `x + 1` of the `where`-expression so that it qualifies the defined expression `x`. The resulting expression is

```
(x where x = 1) + 1
```

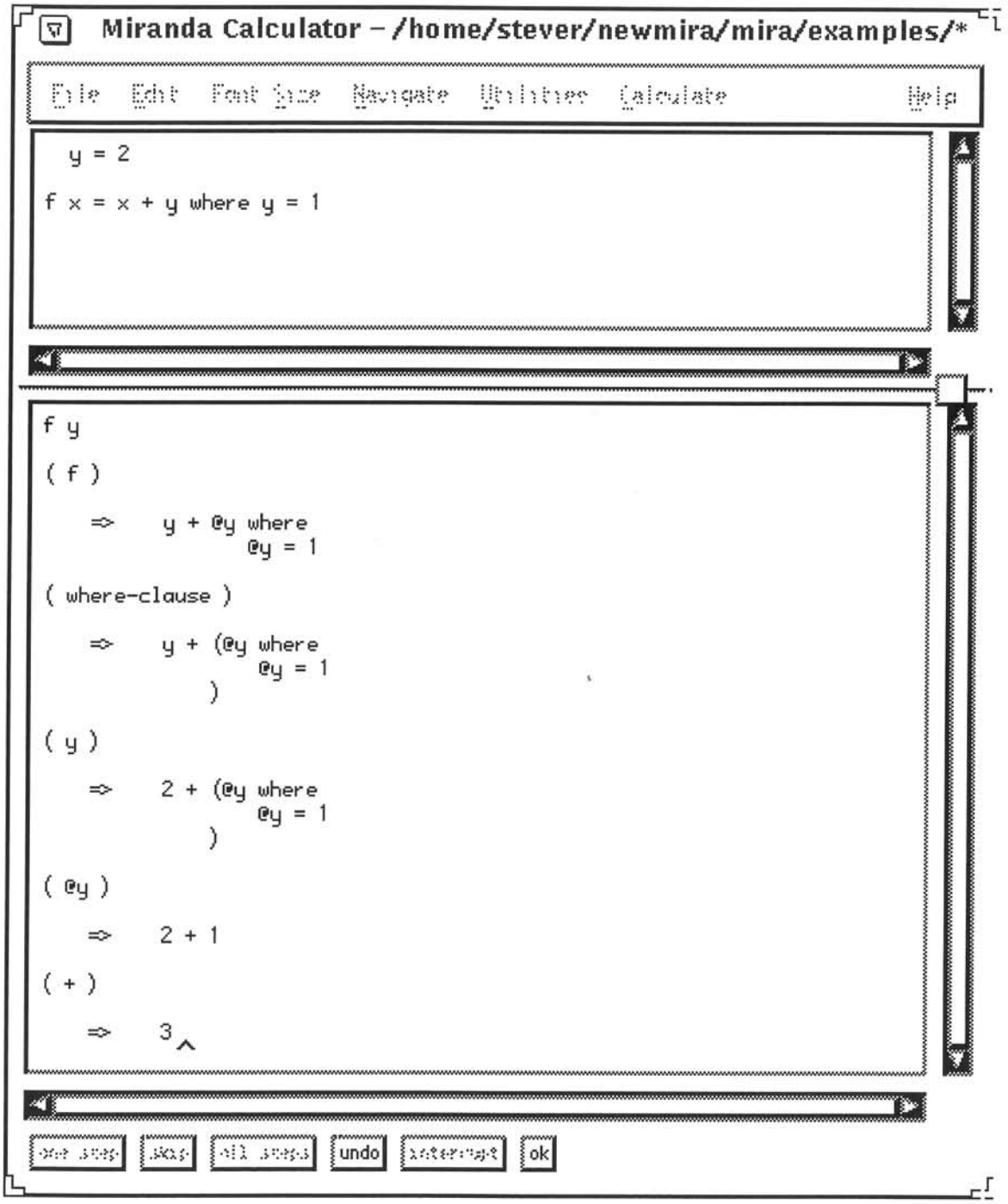
The thing to remember is that this expression, containing an embedded `where`-expression, is *not* a legal expression of Miranda but has been introduced by us in order to show how Miranda `where`-expressions can be evaluated one step at a time.

`where`-expressions have given rise to a second extension to Miranda. Consider the script

```
y = 2
f x = x + y where y = 1
```

What is the value of `f y`? The picture over the page gives the answer. Notice that the first step of the calculation has renamed the definition of `y` in the `where`-clause. This renaming is performed by the rule for evaluating function applications to prevent the capture of names in the function's arguments. If the local definition of `y` (value `1`) in the definition of `f` is not renamed to `@y`, then the argument of `f`, `y` (value `2`), will change its meaning in the course of the calculation. This so-called capture of the free `y` (value `2`) by the binding `y` (value `1`) must be avoided.

The special symbol `@` has been carefully chosen to guarantee that if `x` occurs in a Miranda expression then `@x` does not. The thing to remember is that names beginning with `@` are not legal Miranda names.

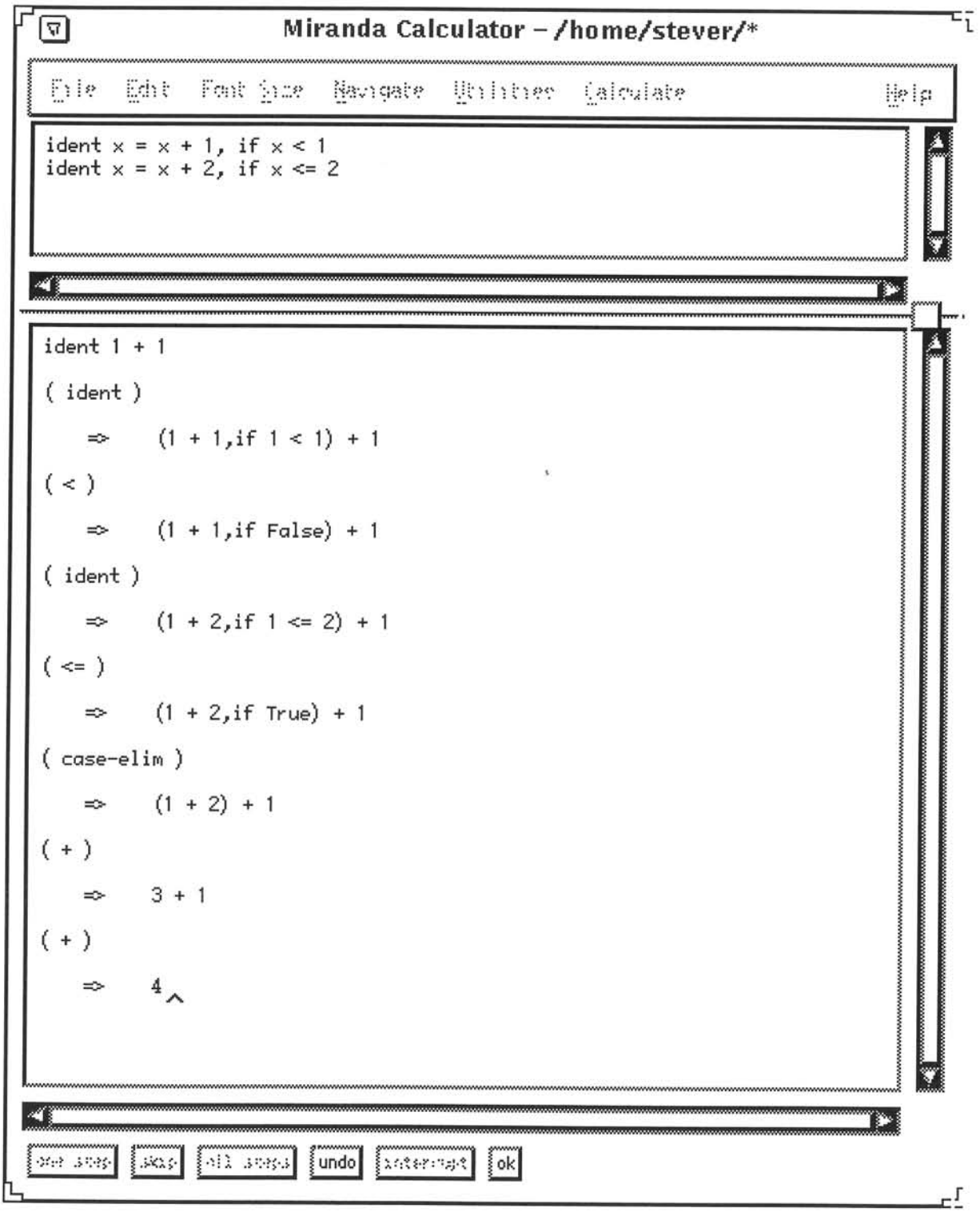

Miranda Calculator – /home/stever/newmira/mira/examples/*
File Edit Font Size Navigate Utilities Calculate Help
y = 2
f x = x + y where y = 1
f y
(f)
=> y + @y where
@y = 1
(where-clause)
=> y + (@y where
@y = 1
)
(y)
=> 2 + (@y where
@y = 1
)
(@y)
=> 2 + 1
(+)
=> 3
undo
ok

4.2 Embedded `if`-expressions

`if`-expressions, like `where`-expressions, can also become embedded during the course of a calculation. Consider a function `ident`

```
ident x = x+1, if x < 1
ident x = x+2, if x <= 2
```

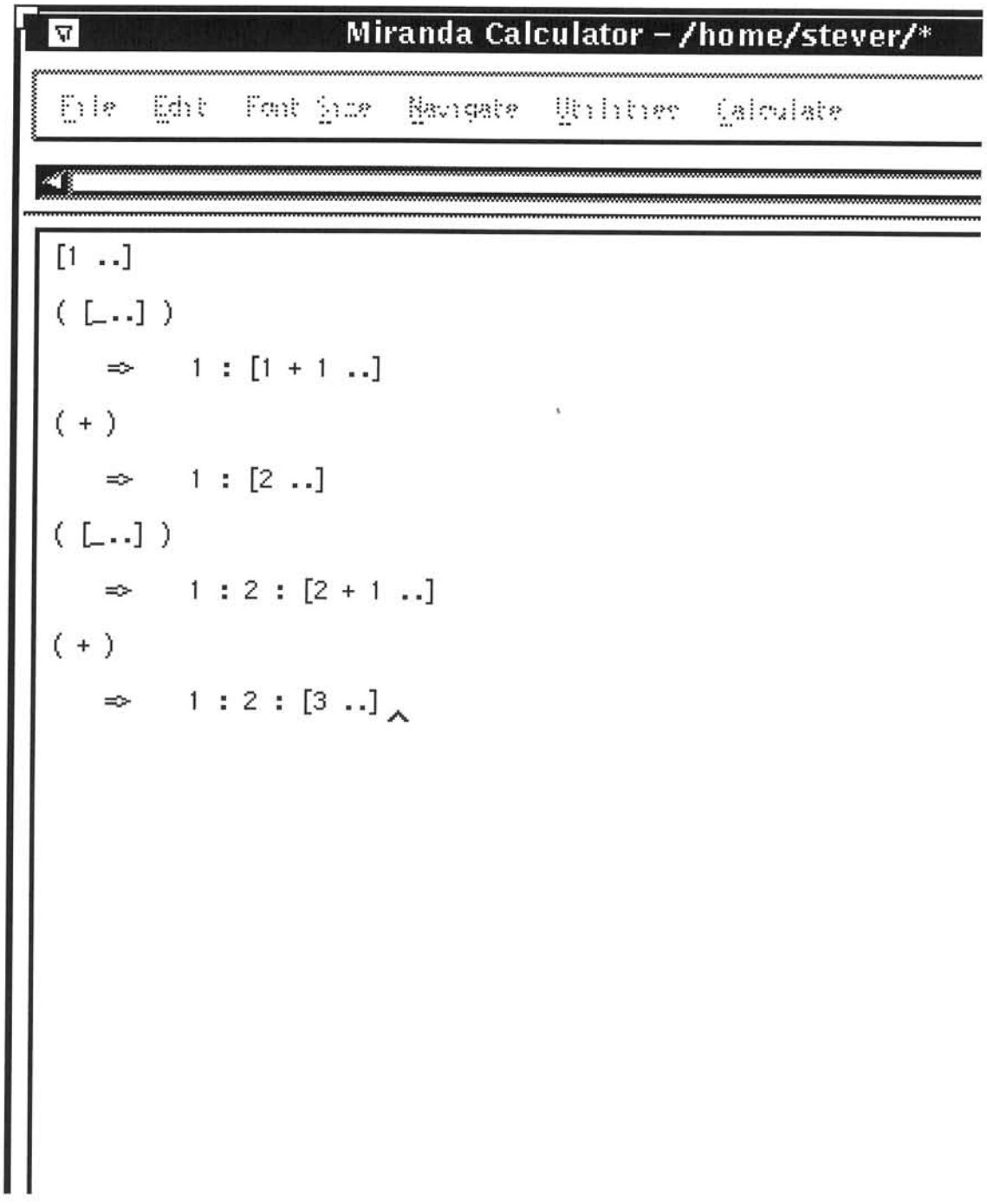
and the calculation of `ident 1 + 1`

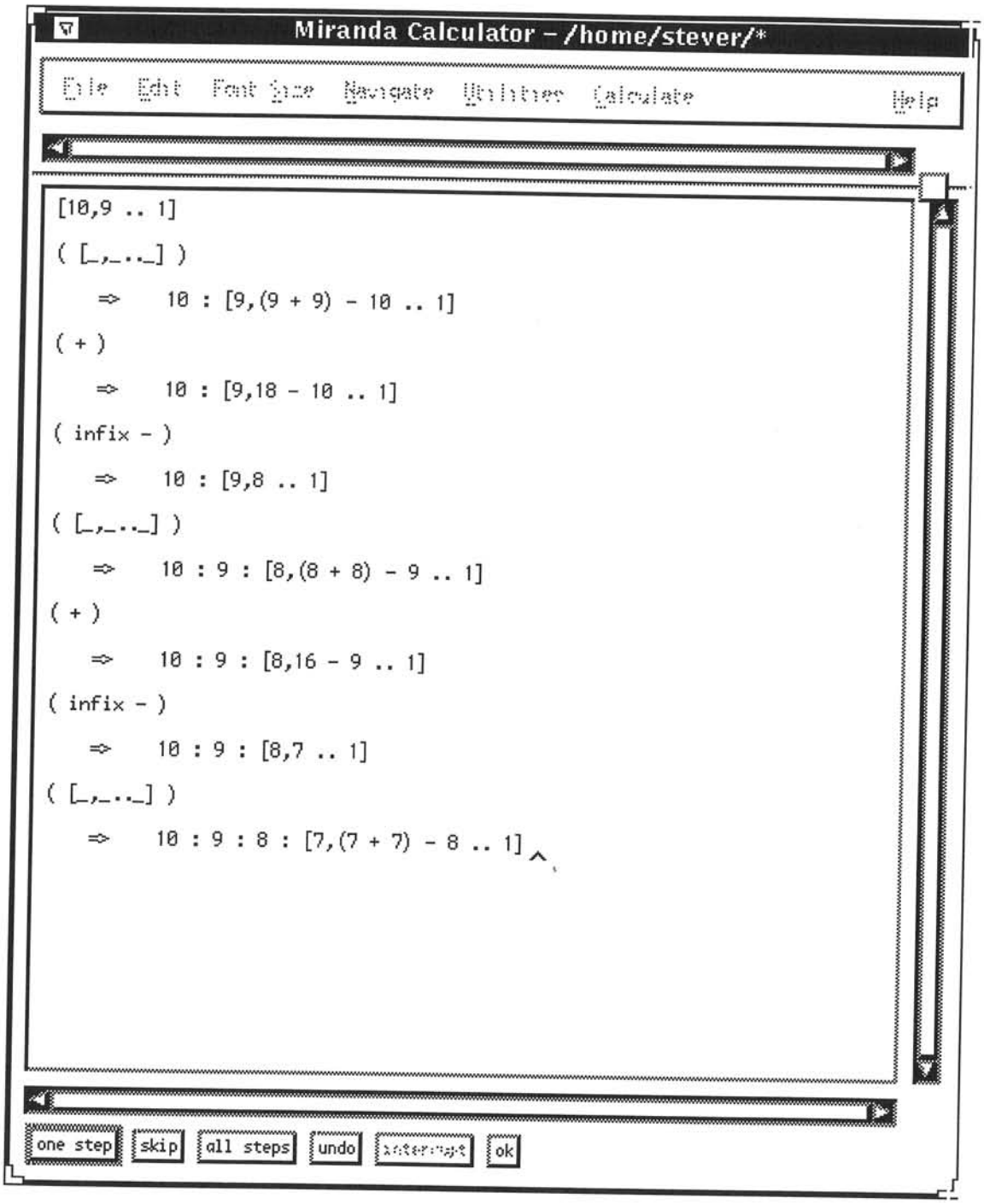


4.3 Iterative expressions: list comprehensions

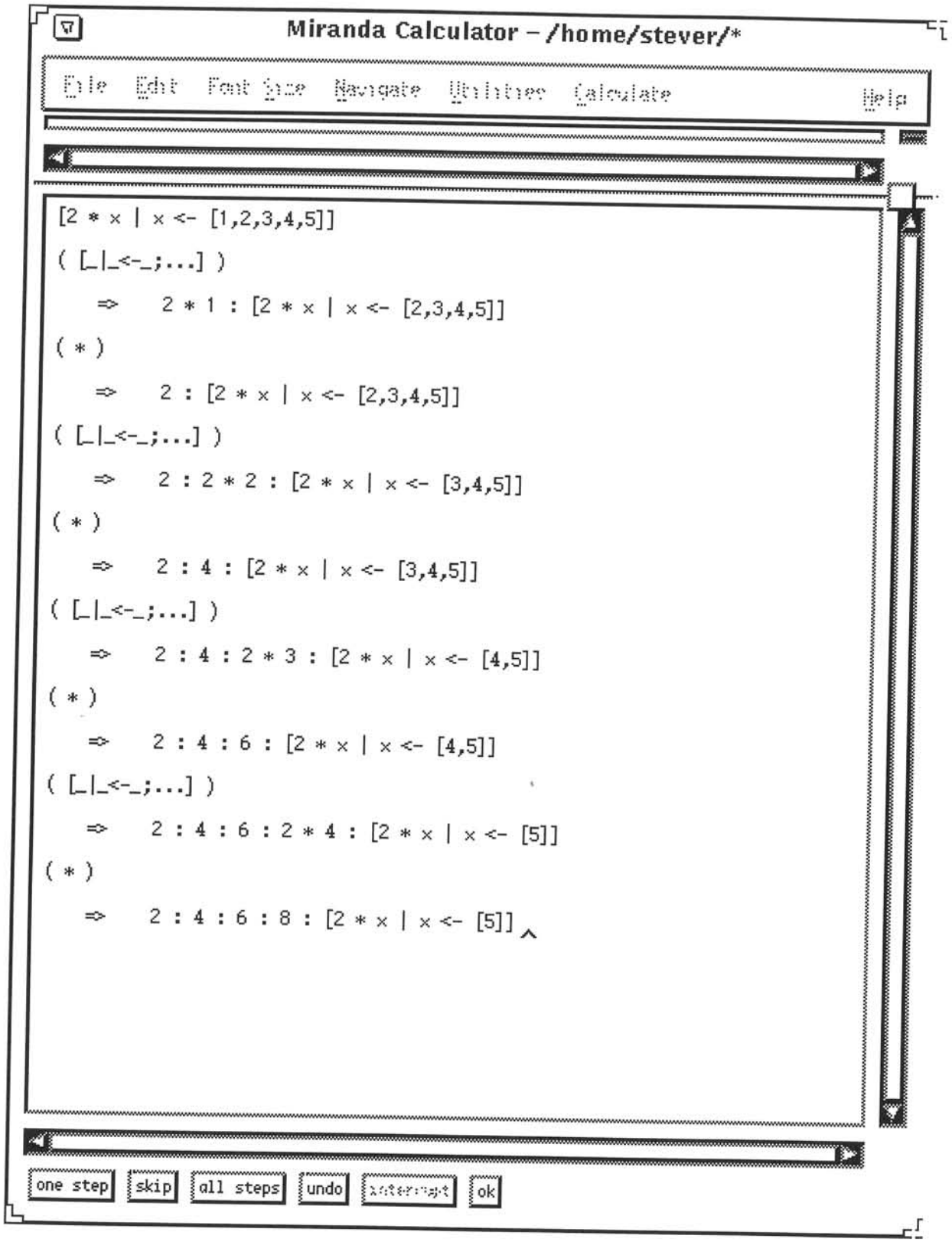
Expressions like `[1..]` and `[10,9..1]` are evaluated by following the rules that define them, and no further additions to the language had to be made in order to explain their evaluation step-by-step.

`[1..]` is evaluated by following the rule that this list is the one defined by the equation `[a..] = a:[a+1..]`, where we consider `[ ..]` as the "name" of a function. Similarly with `[10,9..1]` where the function being evaluated has the name `[_,_.._]`. This is the list whose head is `10`, whose next element is `9` and whose subsequent elements are spaced from `9` by a value equal to the difference of the values of the previous two (i.e. `10 - 9`). The list is continued until `1` is reached. The next two pictures show how these two expressions are evaluated.





More complicated list expressions have required more thought. In the case of a list comprehension like `[2*x | x <- [1,2,3,4,5]]`, the evaluation looks quite straightforward.



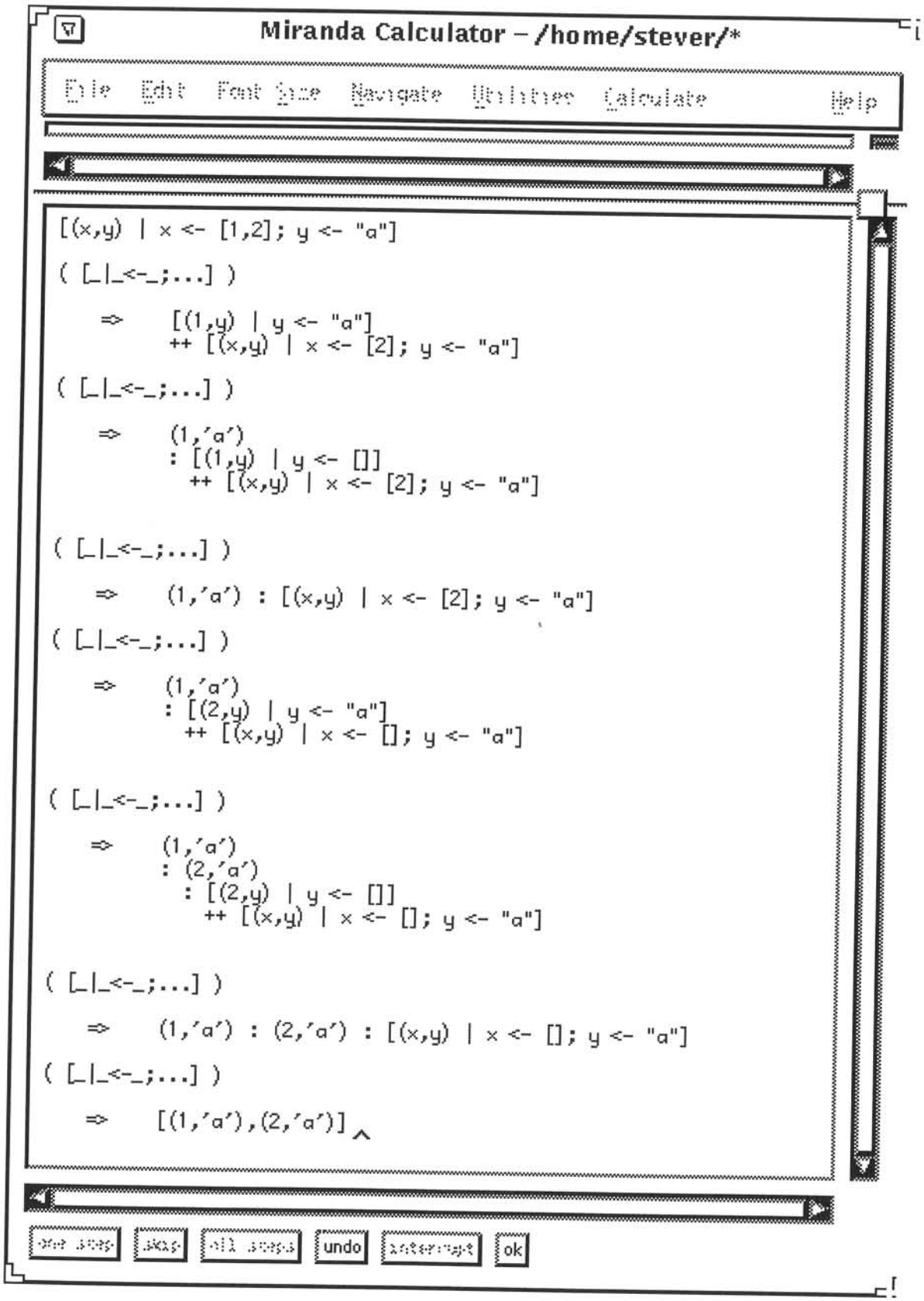
When we start to add further generators and filters then things become slightly more complicated. Note how the evaluation of `[(x,y) | x <- [1,2]; y <- ['a']]` goes. Basically the rule here is that we unravel the iteration by following the structure of the lists, so

```
[(x,y) | x <- [1,2]; y <- ['a']]
```

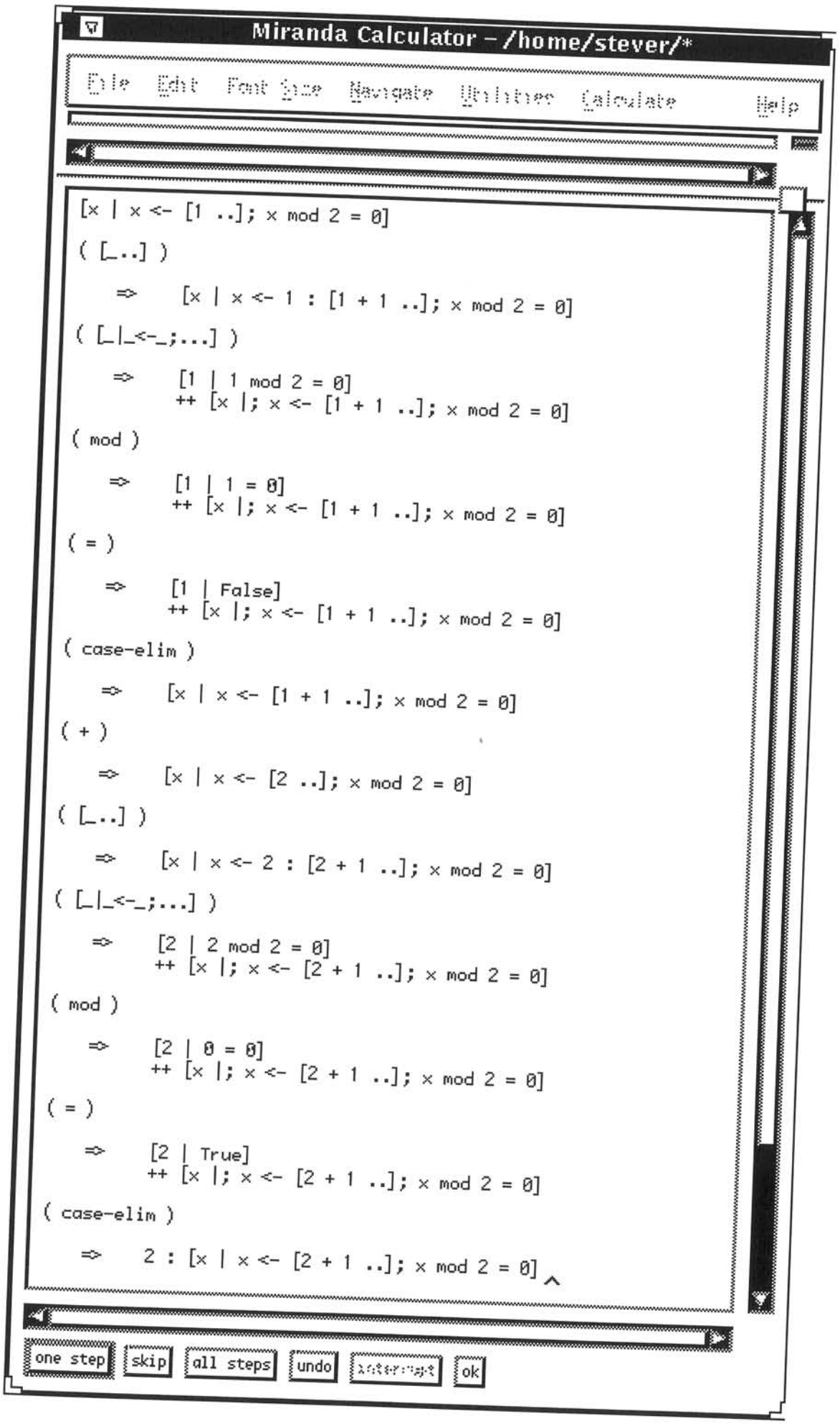
is rewritten to

```
[(1,y) |  y <- ['a']] ++ [(x,y) | x <- [2]; y <- ['a']]
```

so the first list in this concatenation is simple to deal with since it has just one generator, and the second list has been made simpler because one of its generating lists has been made shorter. In this way, in general, the iteration makes progress.



A similar tactic is used in the evaluation of `[x | x <- [1..]; x mod 2 = 0]`.


Miranda Calculator – /home/stever/*
File Edit Font Size Navigate Utilities Calculate Help
[x | x <- [1 ..]; x mod 2 = 0]
([_..])
=> [x | x <- 1 : [1 + 1 ..]; x mod 2 = 0]
([_|_<-_;...])
=> [1 | 1 mod 2 = 0]
++ [x |; x <- [1 + 1 ..]; x mod 2 = 0]
(mod)
=> [1 | 1 = 0]
++ [x |; x <- [1 + 1 ..]; x mod 2 = 0]
(=)
=> [1 | False]
++ [x |; x <- [1 + 1 ..]; x mod 2 = 0]
(case-elim)
=> [x | x <- [1 + 1 ..]; x mod 2 = 0]
(+)
=> [x | x <- [2 ..]; x mod 2 = 0]
([_..])
=> [x | x <- 2 : [2 + 1 ..]; x mod 2 = 0]
([_|_<-_;...])
=> [2 | 2 mod 2 = 0]
++ [x |; x <- [2 + 1 ..]; x mod 2 = 0]
(mod)
=> [2 | 0 = 0]
++ [x |; x <- [2 + 1 ..]; x mod 2 = 0]
(=)
=> [2 | True]
++ [x |; x <- [2 + 1 ..]; x mod 2 = 0]
(case-elim)
=> 2 : [x | x <- [2 + 1 ..]; x mod 2 = 0]
one step
skip
all steps
undo
ok

5. Conclusions

There are other parts of Miranda that we have not discussed in this introduction to *MiraCalc*. These include sections and user-defined types. These are supported by *MiraCalc* and you are encouraged to explore them with its aid.

Please remember that *MiraCalc* is still under development and in particular, while it uses normal-order evaluation it does not support sharing, so it is not truly lazy. Also, the `Skip` and `All Steps` options of the `Evaluate` dialogue are limited in what they can do. However, remember that *MiraCalc* is intended to be used *alongside* Miranda as a support tool when things go wrong. It is *not* intended as a *replacement* . If you try to use it as a replacement you will soon discover its limitations.

MiraCalc was first implemented to run on a Mac (under MacOS and A/UX) and successfully supported first-year teaching of Miranda in the course Functional Programming One during 1992 at QMW, University of London. It is implemented in Prolog (LPA MacProlog in the case of the Mac version and SICStus Prolog with an interface written in C in the case of the Unix version). The Unix version is currently implements more of Miranda than the Mac version, but the Mac version is currently being updated (using LPAProlog32) so that both versions have the same capabilities.

We have no doubt that problems await discovery and it will help us enormously if these are promptly reported to us.