PROVING PROGRAM CORRECTINESS IN LCF

L. Aiello *, M. Aiello **

L74-22

Reprinted from:
Proceedings pg. 59-71, Springer Verlag 1974
(Lecture Notes on Computer Science, Vol. 19)

* Istituto di Elaborazione della Informazione
del C.N.R. - Pisa

** Istituto di Scienze della Informazione,
Università di Pisa
PROVING PROGRAM CORRECTNESS IN LCF

by
Luigia Aiello
Artificial Intelligence Laboratory, Stanford University, California and
Istituto di Elaborazione dell'Informazione, CNR, Pisa, Italy
and
Mario Aiello
Artificial Intelligence Laboratory, Stanford University, California and
Istituto di Scienze dell'Informazione, Universita' di Pisa, Italy

1. Introduction

LCF (Logic for Computable Functions) is an interactive proof-checker. It is based on a
logic (proposed in an unpublished report by D. Scott) where facts about computable
functions are expressible and in which a powerful induction principle (allowing to prove
properties about recursively defined functions) holds. This logic has been augmented in
the implementation of LCF by:

1) a simplification mechanism,
2) the possibility of creating theorems, which can thereafter be used as lemmas,
3) a goal structure with a subgoaling mechanism.

LCF is described in Milner (1972a) (actually the user's manual), Milner (1972b), Milner
and Weyhrauch (1972), Weyhrauch and Milner (1972). Here we can't enter into details
about it, we want only to point out its main applications. Milner and Weyhrauch (1972)
worked out, in LCF, the proof of the correctness of the compiler for a simple
programming language and Weyhrauch and Milner (1972) proved the correctness of a
program for the computation of the factorial function. This program is written in a
programming language in which the basic instructions of input, output and assignment are
expressible as well as the three basic types of "decomposition" (see Dijkstra (1972)),
i.e. concatenation, selection and WHILE-repetition. The syntax and the semantics of this
language have been expressed in LCF: the program for the factorial has been written according to this syntax and its (partial) correctness has been proved, according to this semantics. Furthermore, Newey (1973) provided an environment of axioms for integers, lists, etc. and proved many relative theorems; Newey (1974) proved the correctness of the S-expression form of 'eval'.

The work described in this paper is an extension of Weyhrauch and Milner (1972). The programming language used in Weyhrauch and Milner (1972) is a very simple subset of PASCAL (see Wirth (1971), Hoare and Wirth (1973)). Our main concerns have been to extend that language in order to include many more features of a "real life" programming language and to carry out the correctness proof of some programs written in it. The language described in this paper is also a subset of PASCAL, the main features added to Weyhrauch and Milner (1972) are:

1) the repetition statement REPEAT,
2) procedure calls,
3) declaration of variables,
4) ability of manipulating (inputting, declaring, etc.) arrays.

Two programs written in this programming language have been proved correct. Actually, since the two correctness proofs described in this paper have been completed, the language has been improved and furthermore extended. The two main features of PASCAL not included here are the type definition and the GOTO instruction. The first one, whose treatment is trivial, has been disregarded since we have restricted our attention to functions from integers to integers. As for GOTOs, we have not included them here since we prefer the "sequencing discipline" proposed by Dijkstra (1972), but we have included them in Aiello, Aiello, Weyhrauch (1973) since we wanted to be as close as possible to PASCAL. As one can easily imagine the semantics of a program containing GOTO's are more complex than the semantics of a GOTO-free program, and
this complexity is reflected in the proof of its correctness. However, we want to point out that the semantic functions described in Aiello, Aiello, Weyhrauch (1973) if applied to GOTO-free programs, behave as the semantic functions described here.

2. The Axiomatisation of Syntax and Semantics

For the sake of brevity, we can’t describe in detail all the LCF axioms which define the language. Here we present just few explaining examples. However, in the appendix all the axioms describing the semantic functions are listed. We begin with some example of axioms describing the abstract syntax of the language. A constructor and one or more selector functions are associated with each instruction. For instance:

\[ \forall p_1 \ p_2. \ \text{firstof}(\text{mkcmpnd}(p_1, p_2)) = p_1, \]
\[ \forall p_1 \ p_2. \ \text{secondof}(\text{mkcmpnd}(p_1, p_2)) = p_2, \]
\[ \forall n. \ \text{namof}(\text{mkvardecl}(n)) = n, \]

define respectively the constructor and selector functions for the concatenation of instructions and the variable declaration. A type is associated with each instruction and, for each type, a predicate is defined, which is true only for arguments of that type.

Going back to the previous example:

\[ \forall p_1 \ p_2. \ \text{type}(\text{mkcmpnd}(p_1, p_2)) = \_\text{CM}, \]
\[ \forall n. \ \text{type}(\text{mkvardecl}(n)) = \_\text{VD}, \]
\[ \forall x. \ \text{iscmpnd}(x) = \text{type}(x) = \_\text{CM}, \]
\[ \forall x. \ \text{isvardecl}(x) = \text{type}(x) = \_\text{VD}, \]

The fact that these types are distinct is guaranteed by axioms of the form:

\[ \_\text{CM} = \_\text{VD} \neq \text{FF} \]

Note that, in LCF, true, false and undefined are denoted by TT, FF and UU.

A similar syntax is defined for boolean and arithmetic expressions. While in Weyhrauch and Milner (1972) only binary operators are dealt with, we have extended the language in order to have unary and binary operators. The arity of operators is established by predicates (e.g. binary(plus) = TT, unary(minus1) = TT).
The notion of semantics is based on the ideas expressed in Scott and Strachey (1971) and Weyhrauch and Milner (1972), but we have generalized the notion of storage. In our axiomatization, the storage is not a function from names into values, but from the cartesian product $N \times \text{'names} \times N'$ into values, where $N$ denotes the natural numbers, $N'$ the union of $N$ and a finite set of special symbols:

$$s : N \times \text{'names} \times N' \rightarrow \text{values}$$

The first integer parameter is added in order to determine the environment for the execution of the various procedures, the third parameter specifies whether the "memory location" determined by the name (appearing as second element of the 3-tuple) is a variable, an array element or a formal parameter of a procedure (in this case the corresponding value is the actual parameter).

The semantics of a program $p$ is defined by means of a function $MS$, which, applied to $p$, in the environment determined by the level number 0, maps states into states. The state on which $MS(p,0)$ operates is built up by the input function. The result of the program is furnished by an output function which, when applied to a name, maps states into values. The input and output functions are not part of the program and their meaning is not included in $MS$. It has to be defined according to the particular input required by the program and to the output produced by it.

To prove a program $p$ correct, with respect to a certain function $f$, means that, for all admissible arguments, the result obtained by building up the initial state (by inputting these arguments), then applying to it the function $MS(p,0)$ (representing the semantics of the program), and finally outputing from the final state the value associated to the output variable, is the same as the result of the application of $f$ to them. Suppose that the input to the program is a list of numerical constants, then by introducing a VALUE function:

$$\text{VALUE} = [\lambda p \times \text{isprog}(p) \rightarrow \text{isarglist}(x) \rightarrow$$
$$\text{MO(outvarof}(p), \text{MS(bodyof}(p), 0, \text{MI(inpvarof}(p), x, \text{UU}))), \text{UU}, \text{UU}]$$
we have that correctness can be expressed by:

\[ \forall \text{args. admissible(args)}:: \text{VALUE}(p, \text{INPUT}(\text{args})) = f(\text{args}), \]

where the four dots notation means that if the predicate on the left is true, then the equality on the right is true. MO defines the meaning of the output, MI is the input function for numerical constants. It builds up the initial state starting from the input variables of \( p \) and the values furnished by the assembly function INPUT. In this way it initializes the input variables by both declaring them and assigning them a value. When the input is a vector a \( V \text{VALUE} \) function as well as \( \text{INPUT}V \) and \( \text{MVI} \) are defined in an analogous way (see appendix).

The definition of \( MS \) is:

\[
\begin{align*}
\text{MS} &= [\alpha M (\lambda x. 1) v. \\
& \text{iscmpnd}(p) \rightarrow \text{CMPND}(M(\text{firstof}(p), v), M(\text{secondof}(p), v)), \\
& \text{isvardec}(p) \rightarrow \text{CREA}(\text{namof}(p), v), \\
& \text{isaredec}(p) \rightarrow \text{CREA}(\text{namof}(p), \text{ubofof}(p), v), \\
& \text{isasass}(p) \rightarrow \text{ASSIGN}(\text{lhsbsof}(p), v, M \text{EXPR}(\text{rhsbsof}(p), v)), \\
& \text{iscond}(p) \rightarrow \text{COND}(M \text{EXPR}(\text{thenof}(p), v), M(\text{thenof}(p), v), M(\text{elseifof}(p), v)), \\
& \text{isrepeator}(p) \rightarrow \text{REPEAT}(M(\text{bodyofof}(p), v), M \text{EXPR}(\text{testof}(p), v)), \\
& \text{iswhile}(p) \rightarrow \text{WHILE}(M \text{EXPR}(\text{testof}(p), v), M(\text{bodyofof}(p), v)), \\
& \text{isproccall}(p) \rightarrow \text{MB}(\text{succ}(v), \text{arglistof}(\text{namof}(p)), \text{actarglistof}(p)) \\
& \quad M(\text{bodyofof}(\text{namof}(p)), \text{succ}(v)) \rightarrow \\
& \quad \text{CLEAR}(\text{succ}(v), \text{UU})].
\end{align*}
\]

In the above definition \( \alpha \) denotes the minimal fixpoint operator. MS is defined by means of auxiliary functions, each of which defines the semantics of a particular instruction. For example, if MS is applied to a mkcmpnd, i.e. if the predicate iscmpnd is true, the result is the function CMPND applied to the result of applying MS to each one of the components of the mkcmpnd. The function CMPND is the composition operator and is defined as follows:

\[
\text{CMPND} = \circ, \\
\circ = (\lambda f x. g(f(x))).
\]

We can't enter into details about all the semantic functions, we want only to outline how
some of them are defined. Let us consider the semantics of the procedure call: when a
procedure is invoked in an environment specified by a level number, the value of that
level number is increased by 1 and a stack is built up, by the function MB, which
specifies how the formal parameters are bound to the actual parameters. Then the
function MS is applied, recursively to the procedure body in the environment specified
by the new level number and, finally, the function CLEAR is invoked. This gets rid of the
binding stack and all of the variables local to that procedure call. Note that in the
version of the language used here, value parameters, function and procedure parameters
are not allowed as well as global variables, they have been included in the version of
the language (described in Aiello, Aiello, Weyhrauch (1973)). Finally note that a different
approach has been followed for what concerns procedure and function declarations.
While in Aiello, Aiello, Weyhrauch (1973) a procedure or function declaration is
"interpreted" by MS as any other statement and the procedure or or function is stored
into the memory, here only procedures are dealt with and they are declared apart from
programs. They are "compiled" separately (i.e. the program and the procedure
declaration are two different LCF axioms) and are substituted for their names within the
program when a procedure call is executed.
We conclude the description of the semantics with few comments about the ASSIGN and
MEXPR functions. The result o' the function ASSIGN applied to the 4-tuple (n,v,v',s) is
the state s' obtained from the state s, by assigning the value v to n, in the environment
determined by the level number lv. First a test is done for deciding whether n is a
variable name or an array element, then the state is searched. If n has been declared in
lv, then the assignment takes place, otherwise the binding stack is searched. If n is found
in the binding stack, then ASSIGN applies to the associated name, the predecessor of lv,
v, s. In all the other cases the result is undefined. Finally, MEXPR, the evaluation function for arithmetic expressions, can be informally described as follows: if the expression to be evaluated contains an arithmetic operator, it is applied to the result of the application of MEXPR to its argument(s), otherwise, if the argument of MEXPR is a variable name or an array element, the corresponding value is "read" out of the state with a mechanism similar to that of the ASSIGN function.

3. Two sample proofs

Two proofs of program correctness have been carried out using the previous described syntactic and semantic axioms (see the appendix for the flow charts, and the lists of LCF commands and ask the authors for the printouts of the proof). Actually only the partial correctness has been proved, i.e. ⊢ (less defined) instead of = (equivalent) in the definition of the goal to be proved, the converse being analogous. The two sample programs are:

1) a program which computes the greatest common divisor of two positive numbers with the euclidean algorithm (see, for instance Knuth (1968)),
2) a program which computes the norm of a vector of any length.

The first program was chosen to point out the usefulness of declarations and the mechanism of procedure calls, the second one to demonstrate how arrays are manipulated. The possibility of declaring variables (and arrays) in a program is of great advantage since, otherwise, the only way of creating "memory locations" is by means of the input functions.

We now give the definition of the program EUCL and of the procedure COMPUTE:

EUCL = mkprog(mkinpvar(n1,n2),
    mkrepeate(mkproccall(COMPUTE,mkarglist(n1,n2)),mbexpr1(zero,n2)),
    mkoutvar(n1))

COMPUTE = mkproc(mkarglist(fa1,fa2),
    mckmpnd(mkvardec(n1),
        mckmpnd(mkass(n1,fa2),
            mckmpnd(mkass(fa2,mbexpr2(rmdr,fa1,fa2)))))


mkass(a1,n1)))

This program is proved correct with respect to the function GCD:

\[ \text{GCD} = \text{arg}[\lambda x. y. z.(R(x,y)) - y, g(y,R(x,y))]. \]

In LCF the goal is written as:

\[ \forall x. y. \text{pos}(x)::\text{pos}(y)::\text{VALUE} (\text{EUCLID}, \text{INPUT}(x,y)) \subseteq \text{GCD}(x,y). \]

This means that the goal is proved on the assumption that both arguments are positive.

The program for the computation of the norm of a vector is defined as follows:

\[ \text{NORM} = [\lambda i. x. \text{mkprog}(\text{mkinpvec}(n1)), \text{mkcmpnd}(\text{mkverecl}(n2)), \text{mkcmpnd}(\text{mkverecl}(n3)), \text{mkcmpnd}(\text{mkass}(n2,\text{numeralof}(x))), \text{mkcmpnd}(\text{mkass}(n3,\text{numeralof}(i))), \text{mkcmpnd}(\text{mkwhile}(\text{mkbexpr1}(\text{nonzero}, n3)), \text{mkcmpnd}(\text{mkass}(n2, \text{mkexpr2}(\text{plus}, n2, \text{mkexpr2}(\text{times}, \text{mkae}(n1, n3), \text{mkae}(n1, n3)))), \text{mkass}(n3, \text{mkexpr1}(\text{minus1}, n3))), \text{mkass}(n2, \text{mkexpr1}(\text{sqrt}, n2))))), \text{mkoutvar}(n2)] \]

where mkae(a,i) denotes in the abstract syntax the i-th element of the array a. The correctness of NORM is proved with respect to:

\[ \text{NRM} = [\lambda n. \lambda v. i. r. Z(i) \to \text{SQRT}(r), i \leq n \to g(\text{pred}(i), v, r + (v(i) \times v(i))))(U)] \]

SQRT(r) is an integral approximation of the square root of the number r. The form of the goal to be proved is:

\[ \forall n \forall v s. \text{r.isnalt}(n)::\text{isvector}(n,v)::\text{isnat}(s)::(s \leq n)::\text{isnat}(r)::\text{VALUE}(\text{NORM}(s, r), \text{INPUT}(v, n)) \subseteq \text{NRM}(n, v, s, r) \]

This result is much stronger than the desired one. In fact, we have proved the goal for all vectors v of any length n and for all natural numbers s and n, with s \leq n. The norm of the vector v is computed by setting s=n and r=0. These two extra parameters weren't essential, since the proof could be carried out without them. The same problem was encountered in Weyhrauch and Milner (1972). In the proof of the correctness of the
factorial program a parameter, \( x \), was introduced. Malcom Newey proved the same result without using the parameter, but the proof was much longer. The necessity for extra parameters is quite general: whenever in a program a variable is initialized to a constant value in an environment where a repetition statement is executed and the value of the variable is modified during this execution, it is necessary to substitute for the constant an universally quantified variable ranging over the same domain. In order to prove the correctness of a repetition instruction, an induction is needed and the addition of these extra parameters results in a strengthening of the induction hypothesis.

The two previous programs are very similar from a structural point of view. This is reflected in the structure of the two proofs, which differ just on those parts concerning the \texttt{WHILE} and \texttt{REPEAT} instructions.

4. Conclusions

In this paper we have described a programming language and showed how to use LCF to prove some programs correct. We have given elegant and compact proofs of the correctness of these programs, which although they are still simple, contain more sophisticated programming features than those discussed in Weyhrauch and Milner (1972). We refer to Weyhrauch and Milner for a comparison between this method of defining program semantics and correctness and the other existing methods, as well as for a discussion about the representability of these methods in LCF. In particular, Hoare's formalism (see Hoare and Wirth (1973) and Igarashi, London, Luckham (1973)) is discussed.

Many things remain to be done. First of all a rich base of theorems about the language has to be provided in order to shorten the correctness proofs. Also many improvements can be made to LCF. For instance a proving algorithm which automates the trivial parts
of the proofs. In fact, by looking at the appendix (and to the factorial proof in Weyhrauch and Milner (1972)) it is easily seen how the list of commands is standard: when proving a program correct, all the non-repetition instructions appearing in it have to be evaluated, then an induction has to be done for each repetition instruction occurring in the program, finally the proof ends with a case analysis on the predicate which establishes the exit conditions from the repetition instruction and a proper instantiation of the inductive hypothesis. This instantiation is the only nonstandard part of the proof, even though it is obvious in the two examples shown here.

Acknowledgements

This work has been carried out while the authors have been visitors at the Artificial Intelligence Project of Stanford University. Our sincere gratitude to Malcom Newey and Richard Weyhrauch for useful discussions and to Richard Weyhrauch for introducing us to LCF and reading the manuscript of this paper.

References


APPENDIX

A1. SEMANTIC AXIOMS.

\[ \text{VALUE} = [\lambda p \ x \ x \text{isprog } p \rightarrow \text{isarglist } x \rightarrow \text{M}(\text{outvarof } p, \text{bodyof } p, 0, \text{MS}(\text{inputof } p, x, 0)), \text{M}]) \]

\[ \text{INPUT} = [\lambda x \ y \ x \text{markarglist } (x, y)] \]

\[ \text{MI} = [\lambda x \ y \ s \ x \text{isinputvar } x \rightarrow \text{isarglist } y \rightarrow \text{isname } (\text{firstof } y) \rightarrow \text{isnat } (\text{firstof } y) \rightarrow \text{M}(\text{inputof } x, \text{bodyof } x, \text{isnat } (\text{firstof } y), \text{M}(\text{secondof } x, \text{secondof } y), (x, y)), \text{M})] \]

\[ \text{VVALUE} = [\lambda p \ x \ x \text{isprog } p \rightarrow \text{isarglist } x \rightarrow \text{M}(\text{outputof } p, \text{bodyof } p, 0, \text{MS}(\text{inputof } p, x, 0)), \text{M})] \]

\[ \text{INPUTV} = [\lambda x \ y \ x \text{markarglist } (x, y)] \]

\[ \text{MVI} = [\lambda x \ y \ x \text{isinputvar } x \rightarrow \text{isarglist } y \rightarrow \text{isnat } (\text{intof } y) \rightarrow \text{isname } (\text{nameof } x) \rightarrow \text{isvector } (\text{intof } y, \text{vectorof } y) \rightarrow \text{M}(\text{inputof } x, \text{bodyof } x, \text{isnat } (\text{intof } y), \text{M}(\text{secondof } x, \text{secondof } y), (x, y)), \text{M})] \]
(i;iInt of y)→vec(y,i,UU),UU,UU,UU,
MO= \[\lambda n s . s(0,\text{natof} n, VR),\]
MS= \[\lambda l, n . \lambda \pi l, v\]
iscmpd p→ CMPND(M(firstof p,l,v),M(secondof p,l,v)),
isvardecl p→ CREA(namof p,l,v),
isardcl p→ CREA(subof p,l,v),
isass p→ ASSIGN(rhs of p,l,v,MEXPR(rhsof p,l,v)),
iscond p→ COND(MEXPR(rhsof p,l,v), M(thenoof p,l,v), M(elseof p,l,v)),
iswhile p→ WHILE(MEXPR(testof p,l,v), M(bodyof p,l,v)),
isrepeat p→ REPEAT(M(bodyof p,l,v), M(EXPR(testof p,l,v)),
isproc call p→ MB(succ l,v,larglist(namof(p)), actarglist(p))\[M(bodyof(namof(p)), succ l,v)\rightarrow \text{CLEAR}(\text{succ} l,v),UU),\]
CMND= ! e,
! = \[\lambda f g r . g(f(r))\],
CREA= \[\lambda n u l e v s l v m i . (m=n)\wedge (l=ev)\wedge (i=VR)\rightarrow \text{lam}, s(lv,mi)\],
CREA= \[\lambda n u l e v s l v m i . (m=n)\wedge (l=ev)\wedge (i=UL)\rightarrow \text{lam}, s(lv,mi)\],
ASSIGN= \[\alpha F . \lambda n l v s .
\text{isname} n \rightarrow D(s(lv,n,VR))\rightarrow [\lambda lev m i . (m=n)\wedge (l=ev)\wedge (i=VR)\rightarrow v(s)\rightarrow s(lv,mei)],
D(s(lv,n,BS))\rightarrow F(s(lv,n,BS), pred lv,v,s)\rightarrow UU,
\text{isea} n \rightarrow D(s(lv,namof n),MEXPR(subof n,lv,s))\rightarrow 
[\lambda lev m i . (m=namof n)\wedge (l=ev)\wedge (i=MEXPR(subof n,lv,s))\rightarrow v(s)\rightarrow s(lv,mei)],
D(s(lv,namof n,BS))\rightarrow F(mkae(s(lv,namof n,BS), subof n), pred lv,v,s)\rightarrow UU,UU],
COND= \[\lambda q f g s . q(s)\rightarrow f(s)\rightarrow g(s)\],
REPEAT= \[\lambda g . \lambda q . \lambda q f . \text{COND}(q, fg(f(q))),[f(D)]\],
WHILE= \[\alpha q . \lambda q f . \text{COND}(q, fg(f(q))),[f(I)]\],
MB= \[\lambda l v f a a l v . \text{BIND}(f(a,a), lv . \text{cond}(l=ev)→ NE, s(lv,mi))\],
BIND= \[\alpha F . \lambda f a a l v . st.l isarglist fa . isarglist aa . isarglist(fstof fa) \rightarrow \text{ismake(fstof aa)\/ visae(fstof aa)} \rightarrow 
M(rmof f, rmof aa, lev,
[\lambda lv m i . (l=ev)\wedge (m=fstof l)\wedge (i=BS)\rightarrow \text{fstof aa}, st(lv,mi)],UU,UU),UU],
isname(a)\rightarrow \text{isname(aa)\/ visae(aa)\rightarrow }
[\lambda lv m i . (l=ev)\wedge (m=aa)\wedge (i=BS)\rightarrow \text{aa}, st(lv,mi)],UU,UU],
CLEAR= \[\lambda lv s l v m i . (l=ev)v, s(lv,mi)]\rightarrow UU,UU,UU],
ID= \[\alpha x . x\],
D= \[\alpha x . (NE=x)\rightarrow FF, TT\],
MBEXPR = \[\alpha F . \lambda x . l v s
\text{isexpr}(e)\rightarrow \text{bunary(pof(e)→ MBO}P(\text{bopof(e)}, \text{isbexpr(bargof(e))→ F(bargof(e), l v, s), MEXPR(bargof of e, l v, s))},
\text{binary(pof(e)→ MBOP2(bopof e), isbexpr(bargof(e))→ F(bargof(e), l v, s), MEXPR(barg2of e, l v, s)), UU, UU],
\text{MEXPR = \[\alpha F . \lambda x . l v s
\text{isname}(e)\rightarrow D(s(lv,e,VR))\rightarrow s(lv,e,VR), D(s(lv, e,BS))\rightarrow M(s(lv, e, BS), pred lv, s), UU,
\text{isae}(e)\rightarrow D(s(lv, namof e), M(subof e, e, s))\rightarrow s(lv, namof e), M(subof e, e, s)),
D(s(lv, namof e, BS))\rightarrow M(mkae(s(lv, namof e, BS), subof e), pred lv, s), UU,
\text{isconst}(e)\rightarrow MCONJ(a),
isexpr(e)\rightarrow \text{unary(pof(e)→ MOP1(opof e, \text{M(argof e, l v, s)}),
\text{binary(pof(e)→ MOP2(opof e, \text{M(argof e, l v, s), M(arg2of e, l v, s)}), UU, UU],
,UU), UU]},
A2. FLOW CHARTS OF THE TWO SAMPLE PROGRAMS.

EUCL
input variables n1, n2

\[\text{call COMPUTE(n1,n2)}\]

\[n2 = 0 \quad \text{?} \quad \text{no}\]

output variable n1

COMPUTE
formal arguments f1, f2
declared variable n1

\[n1 \leftarrow f2\]
\[f2 \leftarrow \text{rmod}(f1, f2)\]
\[f1 \leftarrow n1\]

\[\downarrow \text{return}\]

NORM
input variable n2, declared variables n2, n3

\[n2 \leftarrow x; n3 \leftarrow i\]

\[n3 = 0 \quad \text{?} \quad \text{yes}\]

\[n2 \leftarrow \text{sqrt}(n2)\]

\[n2 \leftarrow n2 + \text{mkae}(n1,n3)\]
\[n3 \leftarrow n3 - 1\]

output variable n2

A3. LIST OF LCF COMMANDS:

Program for the euclidean algorithm

TRY SIMPL;
TRY INDUCT 135;
TRY 1 SPREF;
LABEL INDUCT;
TRY 2 SPREF;
TRY SUBST .HEL^ OCC 2;
TRY SIMPL;
TRY CASES Z(\(x,y\));
TRY 1 SIMPL;
TRY 2 SIMPL;
TRY 3 SIMPL;
APPL .INDUCT,y,R(x,y);
SIMPL -;
TRY;BED;

Program for the norm of a vector.

TRY SIMPL;
TRY INDUCT 167;
TRY 1 SPREF;
LABEL INDUCT;
TRY 2 SPREF;
TRY CASES NZ(s);
TRY 3 SIMPL;
TRY 2 SIMPL;
TRY 1 SIMPL;
APPL .INDUCT,n,v,prec(s),
\[r = \text{ssn-v}(s),UU\]
\[\text{ssn-v}(s),UU\];
SIMPL -;
TRY;BED;