Affirm

Demo Handbook

USC Information Sciences Institute

David H. Thompson, Editor

Version 2.0 - May 8, 1981

Corresponds to ***Affirm*** Version 1.21 (March 23, 1981)

Table of Contents

1. Introduction

This document is meant to be used in conjunction with demos by the PV group, and does not attempt to explain all the basics of ***AFFIRM*** use; instead, it tries to draw together in one handy place the pieces of the reference manual, type library, and other documentation we've noticed that most users need, but couldn't quickly find. The document contains a grammar of the command language, synopses of commands, listings of the types you'll be dealing with in the lesson, and listings of

<< truncated>>

2. Type ElemType

2.1. Notes

This type specification is termed a *minimal specification* because it provides only the most basic definition of the type. The name *ElemType* is declared to be a type name, and the reflexive property of equality for elements of type *ElemType* is specified. Nothing more is said, so that *ElemType* is quite general: no assumptions, no restrictions.

2.2. Type Specification

type ***ElemType***;

declare dummy: ElemType;

axiom dummy=dummy == TRUE;

end {***ElemType***} ;

3. Type SequenceOfElemType

3.1. Notes

In contrast to the minimal specification provided for type *ElemType*, the specification for type *SequenceOfElemType* is quite complex. The interface statements provide the basic domain and range information for the operations of the type. The infix statements tell the system to display certain operators in infix form on output; the user is always free to input expressions involving binary operators in infix, but the system will only produce infix output for those operators so declared in the infix command. The axioms detail the meaning of the operations, by specifying the effect the operations have when applied to the constructors of the type. The constructors of a type are operations producing new values of the type, with the property that any value of the type can be expressed in terms of their functional composition. For example, the two constructors of this type are *NewSequenceOfElemType*, producing a null sequence, and *apr*, "append right", which adds an element to the right end of a sequence. It should be obvious that all sequences can be expressed in terms of these two operations.

RuleLemmas are treated as axioms by the system. The difference is in how the user views them. They are intended to be useful primitive facts about the operations that the user wants included in the automatically-applied rewrite rules of the data type. But they are not axiomatic. Rather, they can be proved from the axioms.

Define statements provide "macros" which will be expanded only upon user request. This provides an expression encapsulation facility. Since axioms are turned into rewrite rules that are always applied to an expression during a proof attempt, any axiom that would cause infinite rewrites (such as a statement of operator commutativity) is not allowed. Instead the user can provide the information using a definition.

Finally, schemas provide a means of case analysis or induction, usually structural induction based on the constructors of the data type. We'll say more on this during the demo.

3.2. Intuitive Meaning of the Operators

All operations always accept *values* and return *values*. The operands are not variables in the programming-language sense; operations can't modify their parameters.

Operator	Intuitive Meaning
NewSequenceOfElemType	the empty (null) sequence.
s apr *i*	append element *i* to the right end of sequence *s*.
i apl *s*	append element *i* to the left end of sequence *s*.
s1 join *s2*	concatenate the two sequences *s1* and *s2*. There are several rulelemmas specifying further details of join.
LessFirst(*s*)	"Tail"; all but the first element of the sequence *s*. The axioms don't define the meaning of LessFirst(NewSequenceOfElemType). There's a rulelemma for the apl case.
LessLast(*s*)	all but the last element of the sequence *s*. What's LessLast(NewSequenceOfElemType)? There's a rulelemma specifying the apl case.
dedup(*s*)	removes the duplicate elements from *s*. What do the axioms say about preservation of order?
reverse(*s*)	reverses the order of elements in the sequence *s*.
Rotate(*s*, *k*)	rotates sequence *s* in a left-circular manner, *k* times. *k* is assumed to be a non-negative integer. (Rotate is specified as a definition rather than an axiom.)
Initial(*s*, *k*)	the sequence consisting of the first *k* elements of *s*, if Length(*s*)>*k*, or the sequence *s*, otherwise. (Specified as a definition rather than an axiom.)
LessInitial(*s*, *k*)	the remainder of the sequence *s*, after removing the initial *k* elements. (Specified as a definition rather than an axiom.)
deletepth(*s*, *k*)	removes the k^{th} element from the sequence *s*. What does the definition say about the case *k*>Length(*s*)? Or the case $k \leq 0$?

isNewSequenceOfElemType(s)
: **true** if s = NewSequenceOfElemType, **false** otherwise.

s1 subseq *s2*
: is *s1* a subsequence of *s2*? The definition of subsequence given in the axioms does not require contiguity: <1, 3> is a subsequence of <1, 2, 3>.

NormalForm(s)
: provides a case analysis schema based on the constructors of the type.

Induction(s)
: provides a structural induction schema based on the constructors of the type, NewSequenceOfElemType and apr.

FirstInduction(s)
: provides a structural induction schema based on NewSequenceOfElemType and apl. This schema must be justified.

i in s
: is the element i present in the sequence s? There's a rulelemma covering the apl operation, besides the two axioms.

nodups(s)
: the predicate analogue to dedup(s). There's a rulelemma for the apl case.

disjoint(s, *s1*)
: **false** if s and *s1* have any elements in common, **true** otherwise.

Length(s)
: the number of elements in the sequence s.

First(s)
: the first element of the sequence s. The axioms don't define First(NewSequenceOfElemType). There's a rulelemma for the apl case.

Last(s)
: the last element of the sequence s. The axioms don't specify Last(NewSequenceOfElemType). There's a rulelemma for the apl case.

pth(s,

<<truncated>>

3.3. Type Specification

type ***Sequence***;

needs type Element;

declare s, s1, s2, ss: Sequence;
declare i, i1, i2, ii, j, k: Element;

interfaces
Empty, s apr i, i apl s, seq(i), s1 join s2, LessFirst(s), LessLast(s),
dedup(s), reverse(s): Sequence;

infix apl, apr, join;

interfaces
isNew(s), FirstInduction(s), Induction(s), NormalForm(s),
i in s, nodups(s): Boolean;

infix in;

interfaces
First(s), Last(s): Element;

interface Length(s): Integer;

axioms
s=s == TRUE,
Empty = s apr i == FALSE,
s apr i = Empty == FALSE,
s apr i = s1 apr i1 == ((s=s1) and (i=i1));

axioms
i apl Empty == Empty apr i,
i apl (s apr i1) == (i apl s) apr i1;

axiom seq(i) == Empty apr i;

axioms
Empty join s == s,
(s apr i) join s1 == s join (i apl s1);

axiom LessFirst(s apr i) == if s = Empty
then Empty
else LessFirst(s) apr i;

axiom LessLast(s apr i) == s;

axiom isNew(s) == (s = Empty);

axioms
i in Empty == FALSE,
i in (s apr i1) == (i in s or (i=i1));

axiom First(s apr i) == if s = Empty
then i
else First(s);

axiom Last(s apr i) == i;

axioms
Length(Empty) == 0,
Length(s apr i) == Length(s) + 1;

axioms *The axioms for dedup.*
dedup(Empty) == Empty,
dedup(s apr i) == if i in s
then dedup(s)
else dedup(s) apr i;

axioms *The axioms for reverse.*
reverse(Empty) == Empty,
reverse(s apr i) == i apl reverse(s);

axioms *The axioms for nodups.*
nodups(s apr i) == (nodups(s) and ~(i in s)),
nodups(Empty) == TRUE;

rulelemmas
Empty = i apl s == FALSE,
i apl s = Empty == FALSE;

rulelemmas
s join (s1 apr i) == (s join s1) apr i,
s join Empty == s,
(i apl s1) join s2 == i apl (s1 join s2),
(s join (i apl s1)) join s2 == s join (i apl (s1 join s2)),
s join (s1 join s2) == (s join s1) join s2;

rulelemma LessFirst(i apl s) == s;

rulelemma LessLast(i apl s) == if s = Empty
then Empty
else i apl LessLast(s);

rulelemma i in (i1 apl s) == (i in s or (i=i1));

rulelemma First(i apl s) == i;

```
rulelemma Last(i apl s) == if s = Empty
                           then i
                           else Last(s);

schema
    FirstInduction(s)
      == cases(Prop(Empty),
               all ss, ii (IH(ss) imp Prop(ii apl ss))),

    Induction(s)
```

3.4. Discussion

The lesson is accessed by the command "read lesson.setup;", which loads the needed types and notations and then reads the "theorems" which the learner is to prove. These exercises are somewhat repetitive, but cover the basic set of commands and provide a good feeling for ***Affirm***'s data type induction capability. The user is reminded that not all the "theorems" may actually be such.

3.5. Notation

```
type LessonNotation;

needs types SequenceOfElemType, ElemType;

declare dummy: LessonNotation;
declare s, s1, s2: SequenceOfElemType;
declare i, j, k: ElemType;

interfaces
    deleteNonp(s), dedup(s), reverse(s): SequenceOfElemType;

interfaces
    nodups(s), s1 subseq s2, p(i), allp(s): Boolean;

infix subseq;

axiom dummy=dummy == TRUE;

axioms
    deleteNonp(NewSequenceOfElemType) == NewSequenceOfElemType,
    deleteNonp(s apr i)
      == if p(i)
```

```
            then deleteNonp(s) apr i
            else deleteNonp(s);

axioms
    dedup(NewSequenceOfElemType) == NewSequenceOfElemType,
    dedup(s apr i)
      == if i in s
           then dedup(s)
           else dedup(s) apr i;

axioms
    reverse(NewSequenceOfElemType) == NewSequenceOfElemType,
    reverse(s apr i) == i apl reverse(s);

axioms
    nodups(s apr i) == (nodups(s) and ~(i in s)),
    nodups(NewSequenceOfElemType) == TRUE;

axioms
    s subseq NewSequenceOfElemType == (s = NewSequenceOfElemType),
    s1 subseq (s apr i)
      == (    (s1 = NewSequenceOfElemType) or s1 subseq s
          or LessLast(s1) subseq s and (Last(s1) = i));

axioms
    allp(NewSequenceOfElemType) == TRUE,
    allp(s apr i) == (p(i) and allp(s));

end {LessonNotation} ;
```

3.6. "Theorems"

The following propositions may not all be theorems; that's part of what the lesson is teaching.

theorem AllpDeNonp, allp(deleteNonp(s));
theorem NodupsDedup, nodups(dedup(s));
theorem DeNonpSubseq, deleteNonp(s) subseq s;
theorem DedupSubseq, dedup(s) subseq s;
theorem AllpDedup, allp(s) imp allp(dedup(s));
theorem NodupsDeNonp, nodups(s) imp nodups(deleteNonp(s));
theorem DeNonpJoin, deleteNonp(s1 join s2)=
deleteNonp(s1) join deleteNonp(s2);
theorem AllpJoin, allp(s1 join s2) eqv allp(s1) and allp()

<<truncated here>>

Appendix I
The Syntax of User Commands

The grammatical presentation method used here was designed by David Wile [Wile79a]. In this scheme, terminal symbols are prefixed with a single quote, and are displayed in a **typewriter-like font**. Nonterminal symbols are simple identifiers, and are displayed in *italics*. The form

symbol1 ^ *symbol2*

means

One or more occurrences of *symbol1*, separated by *symbol2*.

For example,

id ^ ',

represents a list of identifiers separated by commas. The form

[*symbolSequence*]

means

Zero or one occurrences of *symbolSequence*.

The empty string is denoted by ϵ, the Greek letter epsilon.

Commands most likely to be needed by an inexperienced user are marked in the left margin with the symbol "**".

Other conventions should be obvious.

```
AffirmCommand := ';
        | '@ [ arbitraryTextExceptSemicolon ] ';

**      | 'abort ';
        | 'adopt typeName ';
**      | 'annotate [ nodeName ', ] arbitraryTextExceptSemicolon ';
**      | 'apply [ nodeName ', ] proposition ';
        | 'arc arcLabel ';
        | 'assume [ nodeName ', ] proposition ';
        | 'augment proposition ';
**      | 'axiom rule ';
        | 'axioms rule ^ ', ';

        | 'cases ';
        | 'choose number ^ ', ';
        | 'clear 'proof ';
        | 'compile objects ';
        | 'complete ';

**      | 'declare id ^ ', ': typeName ';
**      | 'define rule ^ ', ';
        | 'denote ( expression 'by variable ) ^ ', ';
        | 'discard objects ';
        | 'down child ';

        | 'e InterlispCommand
**      | 'edit typeName ';
**      | 'employ schemaName '( allVariable ') ';
**      | 'end ';
        | 'eval expression ';
        | 'exec ';

**      | 'fix [ eventSpecification ] ';
**      | 'freeze [ fileName ] ';

        | 'genvcs procedureName ^ ', ';
**      | 'gripe shortTitle ';

        | 'infix interfaceName ^ ', ';
**      | 'interface lhs ': typeName ';
        | 'interfaces lhs ^ ', ': typeName ';
**      | 'invoke rangedExp ^ ', ';

        | 'let instantiation ^ ', ';
        | 'lisp ';
```

```
        | 'load [ fileName ] ';

        | 'name nodeName [ ', proposition ] ';
**      | 'needs objects ';
**      | 'next ';
        | 'normalize ';
        | 'normint ';
**      | 'note arbitraryTextExceptSemicolon ';

**      | 'ok ';

**      | 'print printOptions ';
**      | 'profile [ transaction ^ ', ] ';
**      | 'put instantiation ^ ', ';

**      | 'quit ';

**      | 'read [ fileName ] ';
        | 'readp [ fileName ] ';
        | 'redo [ eventSpecification ] ';
**      | 'replace [ expression ^ ', ] ';
        | 'resume ';
        | 'retry ';
        | 'review ';
**      | 'rulelemma rule ';
        | 'rulelemmas rule ^ ', ';

**      | 'save objects ';
**      | 'schema rule ';
        | 'schemas rule ^ ', ';
**      | 'search ';
        | 'set variableName 'to expression ';
        | 'stop ';
        | 'storage ( 'normal | 'severe | 'tight ) ';
        | 'sufficient? [ typeName ] ';
**      | 'suppose [ expression ] ';
        | 'swap rangedExp ^ ', ';

        | 'thaw [ fileName ] ';
        | 'theorem [ nodeName ', ] proposition ';
        | 'transcript [ 'on | 'off | fileName ] ';
**      | 'try [ nodeName ', ] proposition ';
**      | 'type id ';

**      | 'undo [ eventSpecification ] ';
```

```
        | 'up [ number ] ';
        | 'use [ nodeName ', ] proposition ';

        | undocumentedAffirmCommands
        ;

allVariable := id ;

child :=
          arcLabel
        | nodeName
        | ordinalInteger ;

coord :=
          number
        | '- number
        | 'ALL
        | 'LAST
        | 'FIRST ;

definedName := id ;

elementName := id ;

eventSpecification :=
          number
        | AffirmCommandName ;

expression := primary [ infixOp expression ] ;

infixOp :=
          '~ '=       | '! '=
        | '< '=
        | '> '=
        | userDefinedOp ;

instantiation := someVariable '= expression ;

interfaceName := id ;

lhs :=
          interfaceName [ '( expression ^ ', ') ]
        | expression interfaceName expression ;

nodeName :=
```

```
          id
        | number ;

objectName :=
          'AffirmObjects
        | 'Arcs
        | 'Axioms
        | 'Commands
        | 'Definitions
        | 'Directories
        | 'Disconnected
        | 'Files
        | 'FileTypes
        | 'Groups
        | 'HelpTopics
        | 'History
        | 'Interfaces
        | 'Lemmas
        | 'Lhs
        | 'Nodes
        | 'PrintObjects
        | 'ProfileEntries
        | 'Schemas
        | 'Theorems
        | 'TypeParts
        | 'Types
        | 'Variables
        ;

objects := objectName ( elementName | lhs ) ^ ', ;

opOrExpression :=
          expression
        | infixOp
        | prefixOp ;

prefixOp := '-          | 'not
        | userDefinedOp ;

primary :=
          prefixOp [ '( expression ^ ', ') ]
        | variable
        | number
        | prefixOp primary
        | '( expression ')
```

```
        | 'if expression 'then expression [ 'else expression ]
        | quantifier identifier ^ ', '( expression ') ;

printOptions :=
          '?
        | 'assumptions
        | 'BadEquations
        | 'both [ printOptions2 ]
        | 'file fileName
        | 'history
        | 'IH
        | 'known objectName
        | 'named
        | 'names
        | 'next
        | 'original
        | 'proof [ printOptions2 ]
        | 'prop [ printOptions2 ]
        | 'result
        | 'status [ printOptions2 ]
        | 'type typeName [ typeParts ]
        | 'unproven
        | 'uses [ printOptions2 ]
        | 'variable
        | 'variables ;

printOptions2 := [ 'list | 'nolist ] ( 'T | '*
        | 'theorem | 'unproved ) nodeName ;

procedureName := id ;

profileEntryName := id ;

proposition :=
          expression
        | nodeName ;

quantifier :=
          'all
        | 'some ;

range := coord [ ': coord ] ;

rangedExp := opOrExpression [ rangeSpec ] ;
```

rangeSpec := '| *range* ^ ', '| ;

rule := *lhs* '= '= *expression* ;

schemaName := *id* ;

someVariable := *id* ;

transaction := *profileEntryName* [('? | '= *profileValue*)] ;

typeName := *id* ;

typeParts :=
```
          'axiom     | 'axioms
        | 'declare
        | 'define    | 'defn.
        | 'interface | 'interfaces
        | 'needs
        | 'rulelemma | 'rulelemmas
        | 'schema    | 'schemas
        ;
```

*undocumented**Affirm**Commands* :

<<truncated here>>

Appendix II
A Beginner's Subset of Affirm Commands

Not all commands are listed here. Rather, the *most useful* ones are enumerated, using the gross categories *Specification*, *System*, and *Theorem Prover*.

System

abort; Returns from a *lower executive*, aborting the pending command (see the ok command).

exec; Invokes the operating system as a lower fork.

fix; Places the text of a command in a text editor, and re-executes the revised command upon return from the editor.

freeze *fileName*;
Saves the entire state of the current system in file *fileName*.

gripe *file*;
Asks for the text of a message and then sends the message (using the Arpanet) to ISI.

load *file*;
Reads a file containing the internal form of a type specification previously saved using the save command.

needs type *typeNames*;
Causes the system to find and read the type specification for each specified type name.

note *comment*;
The comment facility.

ok; Returns from a *lower executive*, and then executes the pending command (see the abort command).

print *option furtherArguments*;
Prints something. Common *options* are:

theorems *names*;
Lists the indicated theorems.

prop *names*;
Lists the indicated propositions (they do not have to be theorems).

type *typeName*;
Lists the specification of the type.

file *fileName*;
Lists the file.

IH; Lists the current definition of the induction hypothesis IH.

known *objectName*;
Lists the names currently associated with elements of the indicated object class.

proof *theorems*;
Lists the proof trees of the indicated theorems.

profile;
Enters a profile dialogue, where each profile entry is displayed and you have the option of providing a new value. The command can also take parameters: see the description in the command synopses.

quit; Closes the transcript file and returns to the operating system.

read *file*;
Reads a file of commands, executing each. Quite useful for reading the text form of type specifications.

readp *file*;
Reads a file containing Pascal programs, building the internal parsed form (see the genvcs command).

review;
Puts the text of the transcript in a text editor, so you can retrace your steps.

save type *typeNames*;
Saves the internal form of the type specifications, each in its own file. The load command can be used to read the files.

undo *event*;
Undoes the effects of the indicated command.

Specification

adopt *typeName*;
Copies the declared variables from *typeName* into the current type. Useful for establishing proof contexts.

axiom *rule*;
Makes a rewrite rule L → R out of the rule L == R, and adds it to the system's rule set. All rules are applied to expressions during simplification, after each theorem-prover command.

declare *ids*: *typeName*;
Declares the names to be variables of the indicated type.

define *rule*;
Makes a rewrite rule out of the equation. Definitions are not automatically rewritten during simplification; you must explicitly request application using the invoke command.

edit *typeName*;
Opens a new type context for subsequent specification commands. See the type and end commands.

end; Ends the current type context and restores the previous one. Specification commands affect only the current type.

interface *op*(*params*): *typeName*;
Defines the domain and range information for an operation. *Params* are variable names, not type names.

rulelemma *rule*;
Treated just like an axiom.

schema *rule*;
Defines induction and case analysis schemas. See the description in the command synopses.

type *typeName*;
Establishes a new context for subsequent specification commands. If the *typeName* is already declared, it is totally redefined by this command. (But you can always undo this command!)

Theorem Prover

apply *name*, *expression*;
Applies the expression as a lemma to the proposition currently being proven.

cases; Raises embedded "Ifs" by applying the special rule
f(if b then x else y) → if b then f(x) else f(y)

employ *schemaName*;
Uses the schema to perform induction or case analysis.

eval *expression*;
Applies the normalization and simplification process to the expression.

genvcs *PascalUnitNames*;
Generates the verification conditions for the indicated Pascal procedures and functions.

invoke *definitionNames*;
Expands the references to a particular set of operations by replacing the reference with the definition.

name *newName*, *oldName*;
Name *oldName* to be *newName*. If *oldName* is omitted, then the proposition currently being proven is given the new name.

next; Moves to the next unproved part of the proof tree associated with the current theorem, in a fairly natural ordering.

normalize;
Simplifies the current proposition. It is not normally necessary to explicitly invoke this command, because it is automatically performed after each theorem-prover command.

put *existentialVar* = *expression*;
Instantiates existential quantifiers.

replace *expression*;
Performs equality substitutions.

search;
Attempts to find a set of instantiations of existential quantifiers that results in reducing the proposition currently being proven to ***true***.

suppose *expression*;
Breaks the proposition P currently being proven into two propositions:

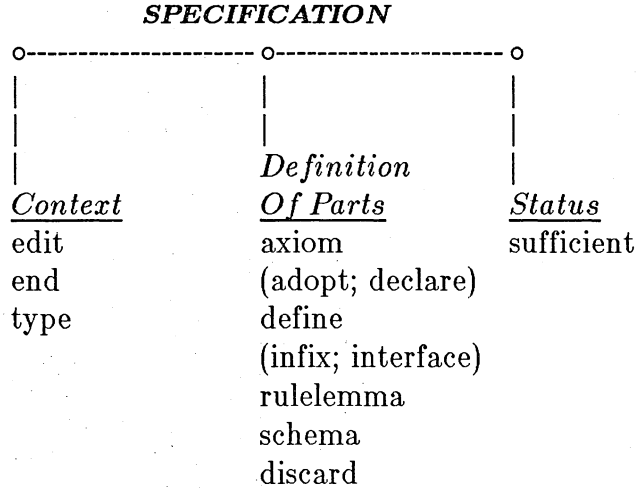
$$expression \supset P$$

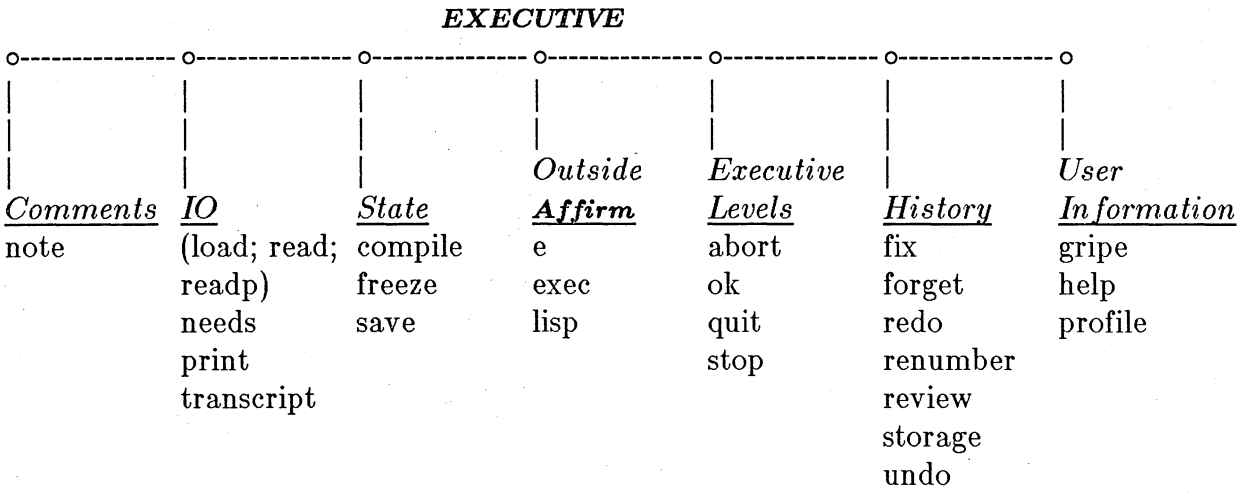
and

$$expression \vee P$$

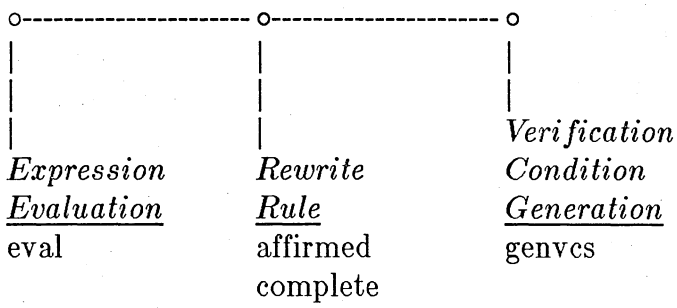
try *name*, *expression*;
Attempts the proof of the named expression.

Appendix III
Command Structure Diagrams

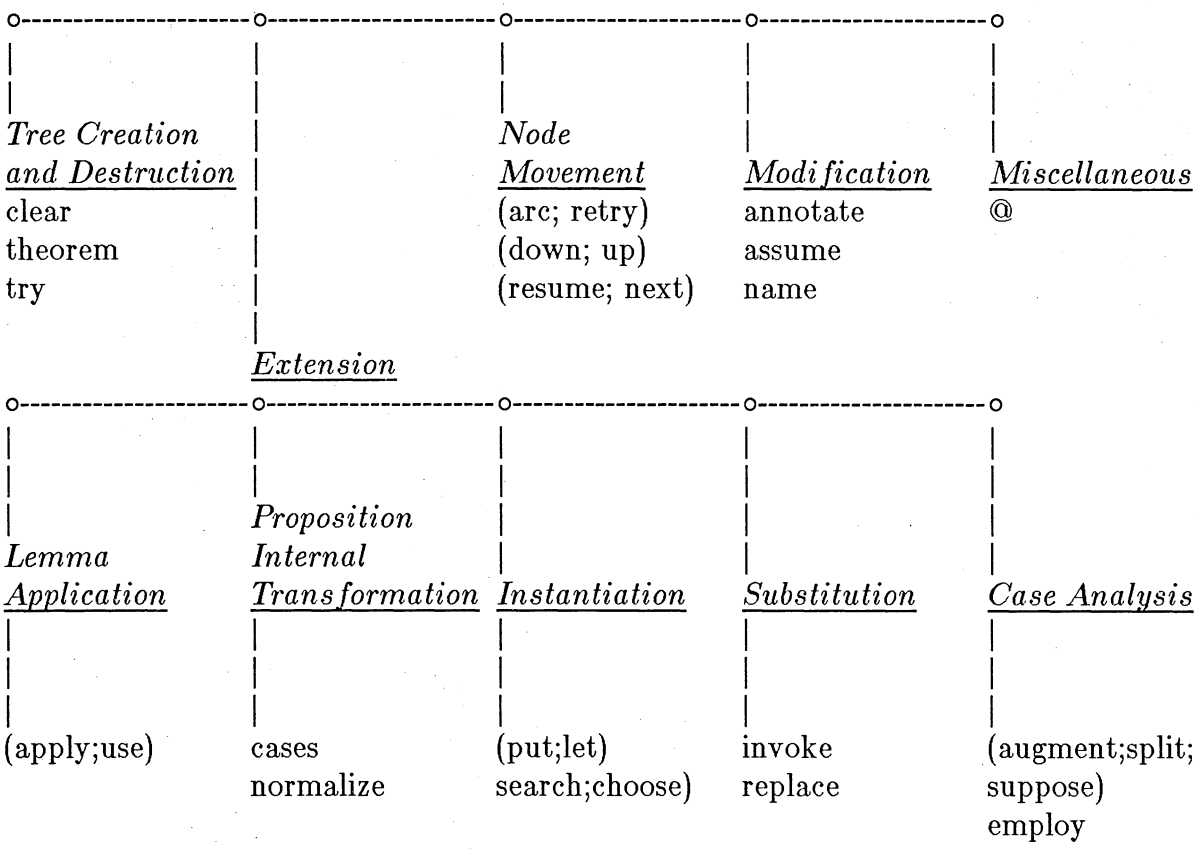




MISCELLANEOUS



THEOREM PROVER



Appendix IV
Command Synopses

This Appendix contains a synopsis of each ***Affirm*** command. For the most part, this synopsis is identical to the description given in the appropriate chapter of the reference manual. The descriptions are gathered here for convenience.

IV.1. Affirm Commands

;

, Semicolon terminates most commands, except for subcommands in the @ (Interlisp editor) and e (escape-to-Interlisp) commands. It may also stand by itself, as a null command.

@ *[annotation];*

, ATCOMMAND Places the user in the Interlisp editor, editing Current Proposition. The annotation is optional, and hopefully documents the less-than-mnemonic Interlisp editor commands that follow INTERLISPEDITOR, IV.3.

abort *;*

Returns the user to the next higher ***Affirm*** executive (if there is one), and aborts the suspended command. The suspended command can then be fixed, or just forgotten.

adopt *typeName;*

Sometimes it is necessary to prove theorems about operators that are associated with types other than the current one. The operators of the type will be referenceable, because the type is in TypeSet. However, the variables of that type will not be referenceable in the current context. Rather than enter the necessary variable declarations manually, the adopt command provides a convenient way to *copy* all the declarations of a type over to the current one. Should any name conflicts occur, the variables being copied will be renamed by appending dollar sign characters ($) to them.

```
adopt SequenceOfElemType;
```

annotate *[proposition,] annotation;*

Attaches a comment to *proposition*; this will appear whenever *proposition* does. *Annotation* is arbitrary text, but cannot contain any semicolons. This is useful for

- documenting where and when an assumption was proven;
- noting what the user's plans are when the proof attempt returns to this spot; and
- commenting a tricky place in a proof.

apply *[nodeName,] proposition;*

This command adds *proposition* to Theorems, and adds it as a hypothesis to the Current Proposition. The command records this dependency by establishing the Uses relationship between the Current Proposition and *proposition*. *proposition* may be assigned a name. The expression corresponding to *proposition* will have its variables renamed to avoid conflicts with variables in the Current Proposition; the renamed form is printed on the terminal. The resultant Current Proposition is not printed,[1] since no meaningful simplification will occur until the user has performed instantiations. Typically, a put command will follow an apply command INSTANTIATION.

arc *arcLabel;*

This command is used to move between cases. Somewhere above Current Proposition is a node with a child labelled *arcLabel*. That child becomes the new Current Proposition. For example, if an induction has three cases (emp:, apr:, and apl:), the user might wish to proceed in an unusual order, saying

```
arc apl:;
...
arc emp:;
...
arc apr:;
```

assume *[nodeName,] proposition;*

Marks *proposition* as *assumed*: it is as if this node were proven (except that this special status is remembered). It may be given a name; this is useful if a file lists assumed facts (such as integer lemmas).

augment *proposition;*

Proposition is added as a hypothesis to the Current Proposition. Separately, the user must show that *proposition* can be deduced from the hypotheses already present. Any free variables in *proposition* are

---

[1] But see the use command USECOMMAND.

identified with those in the Current Proposition, rather than being renamed. Given a Current Proposition of the form

H imp C

this command spawns the two children:

- H imp *proposition*
- (H and *proposition*) imp C

These children are assigned the arc labels thesis: and main:, respectively.

axiom *rule [, ..., rule];*

each *rule* must be an equation *lhs* == *exp*. The rewrite rule *lhs* → *exp* is (normally) added to RuleSet. Variables appearing in *exp* must appear in *lhs*. ***Affirm*** checks all proposed axioms to see how they affect the unique termination of RuleSet. It may interactively simplify the rule, reverse it, or add new rules KNUTH-BENDIX.

```
axioms LessLast(q apr x) == q,
       LessLast(NewSequenceOfElemType) == NewSequenceOfE
       Last(q apr x) == x;
```

cases *;*

Distributes functions over ifs in the Current Proposition.

choose *path;*

Related to the search command, this command allows the user to pick some sequence of instantiations tried by the search command. The search command prints a small integer label to the left of each instantiation it attempts. The sequence of numbers describing the choice--*path*--is the parameter to the choose command. This command is useful if search found lengthy instantiations, but was unable to achieve a final proof.

clear *proof;*

Empties the Proof Forest and Theorems. Erases all proposition names, annotations, and assumptions. Fortunately, this command is undo-able.

compile *type typeName [, ..., typeName];*

Writes a file containing a compiled version of the internal representation of a data type specification (Interlisp code). All stable types should be compiled, since this form of the type uses the least space and runs the fastest. Any types still undergoing development

should be saved, rather than compiled.

complete ;

Attempts to prove the Current Proposition by *reductio ad absurdum* (proof by contradiction). It does this by negating the conclusion of Current Proposition, forming a rewrite rule from it, and (temporarily) adding it to RuleSet. Each hypothesis of Current Proposition is also turned into a rewrite rule and (temporarily) added to RuleSet. The algorithm then tries to generate a contradiction in RuleSet, by performing the unique termination test. If the rule

true → ***false***

is generated, the Current Proposition is proved by contradiction. Otherwise, the final set of rules is used to construct a new result, which may be somewhat simpler than the Current Proposition. This command is sometimes used specifically to rearrange the clauses of a proposition in an inconvenient form.

declare *id [, ..., id]: typeName;*

Each *id* is declared to be a variable of type *typeName*. *typeName* must be a member of TypeSet. Each of the declarations is added to the local declaration set of the current type.

```
declare q, q1: SequenceOfElemType;
declare x: ElemType;
```

define *rule [, ..., rule];*

each *rule* is an equation *lhs* == *exp*. Definitions are rewrite rules, but these rules are only applied when specifically invoked by the user with the invoke command. Definitions are generally used to simplify notation: they are only invoked when needed, so that their contents do not overly complicate propositions. Variables in *exp* must either be bound quantifiers or must appear in the corresponding *lhs*, *but not both*.

discard *disconnected;*

Any nodes which are disconnected (not part of the proof tree of any theorem) are destroyed. Their expressions, annotations, names, and proofs go away. This can save a considerable amount of space. Since the command is undoable, space is only reclaimed when this event is forgotten.

discard *history;*

Purges the history window. This command can be undone.

discard *interface interfaceName [, ..., interfaceName];*

. Discards the indicated operations in the current type. Each operator must be defined in the current type. Note that any references to the discarded operations are inconsistent. The system does not check for this condition. It is the user's responsibility to discard or redefine any rules or propositions referencing the newly-discarded operations.

discard *lhs lhs;*

. *Lhs* must be the left hand side of some axiom, rulelemma, definition, or schema. The rule in RuleSet with left hand side identical to *lhs* is removed from RuleSet. This may destroy the unique termination of RuleSet; no check for this condition is performed.

discard *theorem nodeName [, ..., nodeName];*

. Removes the designated nodes from Theorems. It thus no longer has a proof state, and disappears from summaries of theorems. This is useful when an incorrect lemma has been stated. It is not permissible to discard a lemma which is applied in the proof of some other theorem. If one of the $nodeName_i$ applies another as a lemma, that is okay. ***Affirm*** sorts the list first, and removes the *uses* relationship when the using theorem is deleted. These nodes continue to exist, and retain their proofs; they form part of the disconnected nodes in the tree. The try command will reverse the effects of discard theorem.

discard *variable variableName [, ..., variableName];*

, Discards the indicated variables from the current type. Note that any use of the variables, such as in interface declarations or rules, is now undefined, and may be inconsistent. The system does not presently check for this condition. However, the user will certainly feel the effects later! It is the user's responsibility to discard or redefine interfaces and rules referencing the newly-discarded variables.

denote *expression by variableName [, ..., expression by variableName];*

For each *expression-variableName* pair, this command replaces all occurrences of *expression* with *variableName*, and adds to the Current Proposition the hypothesis

$$expression = variableName$$

down *[child];*

The Current Proposition must have children; this command descends to one of them. *Child* may be:

- an arc label;
- the name of a child;

- an ordinal number (between 1 and the number of children of Current Proposition);
- a node number (if *child* > #children) (*This option is not particularly recommended*); and
- omitted: the first untried child is picked. That failing, the first child is picked.

e *InterlispCommand*] Note that the customary semicolon does not terminate this command. The Interlisp interpreter is invoked with the one command; after the interpreter prints its result, the user is returned to the ***Affirm*** executive.

edit *typeName;*

TypeName must be a member of TypeSet. *typeName* is pushed onto ContextStack, thus making the local declarations of *typeName* available for referencing.

employ *schemaName(var);*

This command permits the use of induction, using any induction schema defined in the relevant abstract data type. *SchemaName* must have been defined (using the schema command) for objects of the same data type as *var*. If the right hand side of the schema definition is of the form

cases(C_1, ..., C_n)

then the resulting propositions $\{C_i\}$ are set up as children of the Current Proposition. The $\{C_i\}$ are expressed in terms of the special predicates Prop(x) and IH(x) which are defined as though the commands

axiom Prop(var) == Current Proposition;
define IH(var) == Current Proposition;

had been given. For example, suppose that Current Proposition, named *SubExtends*, is

sub(q, q1) imp sub(q, q1 apr x)

If the command

employ induce(q);

is performed in the context of

schema induce(q) == cases(Prop(emp),
all q0, x0 (IH(q0) imp Prop(q0 apr x0)));

the following children would result (before simplification):

1. sub(emp, q1) imp sub(emp, q1 apr x)

2. IH(q0, 2 {*SubExtends*[2]})
   and sub(q0 apr x0, q1)
   imp sub(q0 apr x0, q1 apr x)

Quite often, the simplest cases, such as (1), above, normalize to ***true***; the user is notified of these instances. The cursor will automatically be moved to the first nontrivial child.

The cases of an induction are given system-generated labels; these derive from the primary operators underneath Prop in the schema. For example, the above sample would have labels emp: and apr:. Induction is subject to the soundness constraint that *var* must be contained in the all list, and may have no Skolem dependencies upon any variables in the some list SKOLEMIZATION.

`end` *;*

Causes ContextStack to be popped, ending the current type's specification and returning to the previous context.

`eval` *expression;*

Simplifies *expression*, and prints the result. This is useful for testing and demonstrating abstract data types.[3] For more details on its use, see the Users Guide.

`exec` *;*

Invokes the operating system executive as a subroutine. The user can do anything that can be done at the original executive without destroying the files and memory associated with ***Affirm***. To continue with the ***Affirm*** session, the user should type POP at the operating system executive command level.

`fix` *[eventNumber];*

Places the user in a text editor (determined by the profile entry *TextEditor*) with the text of the command issued at event *eventNumber*. The default event when *eventNumber* is not explicitly supplied is the previous event.

---

[2]The two-parameter reference to IH is explained on page @PageIHTWOPARAMETERS IHTWOPARAMETERS.

[3]The user profile entry *ShowRules* SHOWRULESPROFILEENTRY is useful for observing the application of axioms to sample expressions.

freeze *[fileName];*

Causes the entire system state[4] to be written into file *fileName*. The default freeze file name when none is provided in the command is determined by the user profile entry *FreezeFileName*. The size of the file written is on the order of 300 pages. This file can then be run at a later time by simply typing the file name at the operating system executive level. The user will then be back in ***Affirm*** at the executive, as if the freeze had never happened (except that a new transcript file will be opened, if necessary). This command is quite useful for freezing a session in place, and then continuing it later. Compare this with the save command, which does not save the entire system, but just relatively small components of it.

genvcs *procedureName [, ..., procedureName];*

Each *procedureName* must be a Pascal procedure or function[5] unit previously parsed via the readp command. Verification conditions are generated for each *procedureName*.

gripe *subject;*

Creates a message to be sent via the ARPANET to ***Affirm*** maintenance personnel. The system will ask the user to type the body of the message, which is terminated by **control-Z**. After the message is completed, the user has the options of sending the message, or aborting the gripe. The transcript command] can also be sent along as a separate message if it is pertinent to the documentation of the problem or suggestion.

infix *operatorName [, ..., operatorName];*

Each *operatorName* is declared to be an infix operator.

interface *expression [, ..., expression]: typeName;*

Just as declare establishes the types of variables, interface provides the necessary characteristics of operators. All operators should be declared using the interface command before they are referenced in other ***Affirm*** commands. Each *expression* will be an expression of the form $operatorName(var_1, ..., var_m)$, where each of the var_i is a variable declared in the current type. The interface declaration states that *operatorName* is a function of m arguments, with argument types corresponding to those of the var_i. The value returned by

---

[4]The freeze command does not save the state of any open files.

[5]As of ***Affirm*** version 1.21 the verification condition generator did not process functions correctly.

operatorName will be of type *typeName*. In the case of an operator taking no arguments, the parentheses may be omitted. *infix* notation, such as **q apr x**, can also be used.

interface q apr x, apl(q, x): SequenceOfElemType;

invoke *rangedOp [, ..., rangedOp];*

Each of the specified operators should occur in the Current Proposition and have a definition. The definition is expanded. If an operator appears in its own definition, the new occurrence will not be expanded; thus the process will not loop. An ordinal range may be specified; if it is not, the first occurrence of each operator will be expanded. Some examples:

invoke IH;	invoke the first IH
invoke IH |2|;	second IH
invoke IH |all|;	all IH's
invoke IH |-2|;	next to last
invoke IH |2:4|;	second, third and fourth
invoke F(i,j)|2:5|, G|3,5|;	second through fifth occurrences of F(i,j) and the third and fifth occurrences of G

This command can be automatically invoked by the *AutoInvokeIH* profile entry.

let *var=exp [, ..., var=exp];*

This command has the same effect as the put command, except that the new result is the *disjunction* of the unchanged and the instantiated versions of Current Proposition. Thus, all variables in the some list remain subject to further instantiation with the put or let commands. This is useful if the user is not quite sure about an instantiation, or wishes to perform multiple instantiations. It does, however, double the size of the expression. If, for example, the Current Proposition was

all x some y(x): P(x, y)

The command

put y=x;

would give (before simplification)

all x: P(x, x)

while the command

let y=x;

would yield (before simplification)

all x some y(x): P(x, y) or P(x, x)

lisp ;

The ***Interlisp*** interpreter is invoked. The user can next perform any Interlisp command. The OK command (without a semicolon after it) returns the user to the ***Affirm*** executive.

load *[fileName];*

Causes ***Affirm*** to load file *fileName.* The file must have been previously written using the save command. A data type specification is the only ***Affirm*** object that can be saved and then loaded. Note that the file's contents are *not* normal text, and cannot be directly modified by the user.

name *nodeName, [proposition];*

Christens the proposition; the system will henceforth refer to it by the name *nodeName.* If this name is already in use, the system displays its old value.

needs *type[s] typeName [, ..., typeName];*

Should be used immediately after a type command, before any other part of the type specification. This command ensures that each *typeName* are either loaded or read, before any more of the specification of the current type is processed. If the type is already defined, no further processing occurs. If it is not yet defined, then the most recent version of its specification is found. The algorithm which finds the files containing the types to be defined searches a set of directories for the most recent version of the specification of each type, whether that version be in original source form or in the internal saved form, or even in compiled form. For each type requiring such a directory search, ***Affirm*** first identifies the possible set of files containing versions of the type specification; it then ranks the versions (by using the file write date to determine which file was most recently written). ***Affirm*** will normally then proceed to load or read that file, as is appropriate, unless the profile entry *TypeNeeds* is set to Ask. The user will be asked to point out the correct type specification to be input. The set of directories used as of ***Affirm*** version 1.21 is {*Connected*, *Login*, PVLibrary, Affirm}.

next ;

Moves to the next task, according to a depth-first plan, using the following hierarchy:

1. If the Current Theorem has leaves, move to the next one, in a left-to-right ordering of the leaves of the proof tree.

2. If the Current Theorem uses an unproven lemma, try it.

3. If the Current Theorem is used as a lemma by an unproven theorem, return to the theorem. This process extends to any unproven ancestor.

4. If none of the above hold, then stay put and perform the command

    print unproven;

Within this hierarchy, the most-recently-attempted theorem is preferred. Where possible, resume.

normalize *;*

Causes the Current Proposition to be (again) normalized and printed. Since propositions are normalized upon becoming the Current Proposition, this will normally have no effect, but may be necessary due to the occasional incompleteness of the simplification process.

note *arbitraryTextExceptSemicolon;*

The text is placed in the transcript. No other processing is performed.

ok *;*

Returns the user to the next higher ***Affirm*** executive (if there is one), and resumes processing of the suspended command. If this command still has errors in it, the user may well be placed into a lower executive once again. The abort command is useful here, too.

print *?;*

print ?;
displays a list of all the keywords that can follow the command word print. Equivalent to the command

    print known PrintObjects;

print *assumptions;*

Lists all the assumed propositions, and the theorems that depend on them.

print *BadEquations;*

Lists the rules that have been suppressed during the various Knuth-Bendix KNUTH-BENDIX convergence tests, if any.

print *both [list | nolist] whatNodes;*

Like print proof but lists all the propositions in the proof tree. *Verbose.*

`print` *file fileName;*

Copies the contents of file *fileName* to the terminal, and also to the transcript.

`print` *history;*

Prints the user-issued commands still resident in the history window.

`print` *IH;*

Prints the definition of each of the inductive hypotheses IH in the Current Proposition (if any).

`print` *known objectName;*

Enumerates the currently defined set of names in the object class *objectName*. The object names as of ***Affirm*** version 1.21 are AffirmObjects, Arcs, Axioms, Commands, Definitions, Directories, Files, FileTypes, Interfaces, Lemmas, Nodes, PrintObjects, ProfileEntries, Schemas, TypeParts, Types, and Variables.

`print` *[parts typeParts] [types typeName [, ..., typeName]] lhs lhs [, ..., lhs];*

This command provides the rudimentary capability of listing those rules that match some *pattern*. *TypeParts* is a list selected from the set {axiom, lemma, defn, schema}; the list may be empty, in which case the default value axiom is used. *TypeNames* is a list of type names; the list may be empty, in which case the keyword need not be typed. The default value is the list of all currently defined types. *Pattern* is an expression, restricted to one of two simple forms: operator, or operator1(operator2). Each rule in the requested set of types that is a member of one of the requested parts is pattern-matched against the pattern; if it succeeds, the rule is listed. If it fails, the rule is ignored. Only the left-hand side of a rule is used in the pattern-matching process. If the pattern is a simple operator, a match succeeds if the *main* operator of the left-hand side is this operator. If the pattern is of the form operator1(operator2), then operator1 is the main operator, and operator2 is *any internal* operator. If the left-hand side of a rule has operator1 as its main operator, and contains a reference to operator2 as an internal operator, the match succeeds. For example, the command

print lhs join(apr);

will list all the axioms whose left-hand-side main operator is *join*, and which also reference the operator *apr* as an internal operator. This example is useful for the type SequenceOfX, for quite a few types X.

`print` *next;*

This command displays the proposition that the next command would

make the Current Proposition.

print *original;*

Prints the unnormalized form of the Current Proposition (not particularly useful).

print *proof [list | nolist] whatNodes;*

Displays the proof tree. The default for *whatNodes* is `T`. **`List`** causes any lemmas that are used in the proof of Current Theorem to be listed. Note that *whatNodes* does not have to be a theorem, so the user can print a partial proof tree.

print *prop [list | nolist] whatNodes;*

Lists the propositions and their associated names. For example,

```
print prop T;
```

prints Current Theorem.

print *result;*

Prints the Current Proposition THEOREMPROVERSTRUCTURES in its normalized form.

print *status [list | nolist] [whatNodes];*

Tells whether the specified theorems are tried, untried, awaiting lemmas, proved, or assumed. The default when *whatNodes* is omitted is **`theorems`**.

print *type typeName;*

TypeName must be a member of TypeSet. The declarations, needs, interfaces, infix operators, axioms, rulelemmas, definitions, and schemas of type *typeName* are printed on the terminal. Should only a subset of these be desired, *typeName* may be followed with a list of *qualifiers*.

```
print type ElemType;
print type SequenceOfElemType decl schema;
```

print *unproven;*

Prints the status of all unproven theorems.

print *uses [whatNodes];*

Which lemmas are used where? The default for *whatNodes* is **`theorems`**.

print *variables;*

Lists just the *variables* in the Current Proposition; this is useful if the expression is too big to be conveniently displayed as a whole.

`profile` *;*

The profile enquiry dialogue is initiated with the user. The question mark command ? is quite useful here in order to determine what the options are at each step.

`profile` *profileEntryName [= value] [, ..., profileEntryName [= value]];*

Each referenced profile entry is either displayed with its current value or modified, as is appropriate.

`put` *var = exp [, ..., var = exp];*

Each *var* must be a variable in the some list of the Current Proposition. Each *exp* is an expression upon which the corresponding *var* can legally depend SKOLEMIZATION. The *exp* is substituted for the corresponding *var*.

`quit` *;*

Stops ***Affirm***, returning to the operating system executive. The user can return to ***Affirm*** by typing CONTINUE at the operating system executive command level.

`read` *[fileName];*

Causes ***Affirm*** to read *fileName*. The file must contain ***Affirm*** commands. The last command in the file must be the stop command. *FileName* is a text file that the user presumably created using some text editor.

`readp` *[fileName];*

Causes ***Affirm*** to read *fileName*. The file must contain Pascal programs. *FileName* is assumed to be a text file.

`redo` *[eventNumber];*

Re-executes the command at event *eventNumber*.

`replace` *[expression [, ..., expression]];*

If no argument is given, then every hypothesis in Current Proposition of the form L = R is used to replace all other occurrences of L with R. Each *expression* should occur in an equality hypothesis (of the form *expression* = *R* or *R* = *expression*). All other occurrences of *expression* are replaced with *R*. For example, if Current Proposition is

(fee(j, k) and j = m and n = k) imp fie(m, n)

replace; will yield

(fee(***m***, k) and j = m and n = k) imp fie(m, ***k***)

while the command replace m, n; will yield

(fee(j, k) and j = m and n = k) imp fie(***j***, ***k***)

resume *;*

The Current Theorem must be tried. The Current Proposition is restored to the value it had when the user was last proving this theorem, thus resuming a partially-completed proof. The resume command is usually preceded by a try command.

retry *;*

This command is equivalent to the (otherwise unspeakable) command

try Current Theorem;

In other words, this command retries the current theorem.

review *;*

Places the user in a text editor determined by the profile entry *TextEditor*, with the transcript file. The user can then use editor commands to review the events in the file. Each command begins with "U:".

rulelemma *rule [, ..., rule];*

The rulelemma command is a synonym for the axiom command.

save *type typeName [, ..., typeName];*

Causes ***Affirm*** to write files containing the specifications of the indicated types. The file name of each file is the upper-case version of the corresponding type name. The save command can be used in conjunction with the load command to remember data type specifications across ***Affirm*** sessions. The file written by the save command for each type is the internal form of the type specification (Interlisp code). Thus little processing is required to load the type back into ***Affirm***, compared to the processing required when first creating the specification. The file name of the file is obtained by upper-casing the type name; thus type names may not differ only in casing, due to the possible file name conflict.

schema *rule [, ..., rule];*

Each *rule* is an equation *lhs* == *exp*. The schema command introduces induction rules. The soundness of schemas is not determined by ***Affirm***; the user must establish this property. It is in schema declarations that the restriction imposed on equations is most

often felt. The following declaration illustrates a very common error:

```
schema Induction(q) ==
        cases(Prop(NewSequenceOfElemType),
            all q, x(IH(q) imp Prop(q apr x)));          (bad!)
```

Here the parameter is the same identifier as the quantifier in the expression. A correct schema declaration would be:

```
schema Induction(q) ==
        cases(Prop(NewSequenceOfElemType),
            all q0, x(IH(q0) imp Prop(q0 apr x)));
```

search ;

Uses the method of *chaining and narrowing* CHAININGANDNARROWING to attempt to automatically find the instantiations sufficient to reduce Current Proposition to ***true***. The command displays the sets of instantiations it tries. These may be referenced by the user in the choose command.

set *variable to expression;*

Variable no longer represents itself; it is assigned a value which will replace it whenever an expression is normalized. This effect is permanent until *variable* is explicitly given another value. This may be useful in conjunction with the eval command. Other than that, it is not recommended.

stop ;

Should be used only in a file of ***Affirm*** commands, as the last command. It avoids the usual end-of-file problems.

storage *degree;*

Degree is one of {normal, severe, tight}.

sufficient? *[typeName];*

TypeName must be a member of TypeSet. A sufficient-completeness check is performed and the results displayed on the terminal.

suppose *[proposition];*

This command splits the Current Proposition into two children:

- *proposition* imp Current Proposition
- *proposition* or Current Proposition

These children are labelled yes: and no:. If *proposition* is not supplied, the splitting predicate is automatically generated by ***Affirm*** using the

internal If-Then-Else form of Current Proposition. Basically, the predicate is chosen from the first significant branch point. For example, if the Current Proposition is of the form

((A imp B) and H) imp C

the suppose command will yield the two children

A and B and H imp C *and* (~A) and H imp C

the children generated by the suppose command when no explicit proposition is supplied are labelled first:, second:, etc. It usually produces only two. Its detailed description follows, but it is usually best to experiment.

If Current Proposition is of the form:	The children are:
if B then C1 else C2	{B imp C1, B or C2}
C_1 and C_2 and ... and C_k	{C_k, C_1 and ... and C_{k-1}}
H imp (C_1 and ... and C_k)	{H imp C_1, (H and C_1) imp C_2, (H and C_1 and C_2) imp C_3, ..., (H and C_1 and ... and C_{k-1}) imp C
(H1 and (H2 imp C1) and H3) imp C2	(H1 and (~H2) and H3) imp C2}

split *;*

, SUPPOSECOMMAND A synonym for the suppose command with no parameter.[6]

swap *rangedExp [, ..., rangedExp];*

This command reverses equality hypotheses in the Current Proposition. Thus, it is often useful in conjunction with the replace command REPLACECOMMAND. Each *rangedExp* specifies one or more equalities to be reversed. Such a specification may give one of the arguments to the equality, or an ordinal range, or both. For example:

[6]The split command is an obsolete command; its function has been merged into the suppose command.

swap a;	Swap all equations whose left hand side (or right hand side) is the expression *a*.
swap |2|, |-2|;	Swap the second equation, and the next-to-last
swap a |-1|;	Swap the last equation whose left-hand or right-hand side is the expression *a*.

thaw *[fileName];*

This command is the opposite of the freeze command. It takes one parameter, the name of a file containing a session frozen by a freeze command. Most users will not ever have a use for this command, since the frozen session can be started in ***TOPS-20*** or ***Tenex*** simply by typing the file name at the operating system executive level.

theorem *[nodeName,] proposition;*

This command simply enters the proposition into Theorems. It does not affect Current Proposition or Current Theorem. The command creates a root in the Proof Forest that may later be attempted. The user may associate a name with the theorem. This command is especially useful for command files containing lists of theorems to be attempted.

transcript *[fileName];*

Begins a (new) transcript file *fileName*. If there is no transcript file at the time the user issues this command, then the file name of the new transcript, if not provided in the command, is governed by the profile entry *TranscriptFileName*. If there is a transcript file at the time this command is issued, then the new file name, if not provided in the command, is identical to the old file name, with a new version number. The transcript file when the system first begins is written into the user's login directory, rather than the connected directory. Later transcript commands default to the connected directory.

transcript *toggle;*

Toggle is one of {off, on}. This command turns transcript processing either off or on. The file name is determined from the profile entry *TranscriptFileName*.

try *[nodeName,] proposition;*

Makes *proposition* be the Current Proposition. If *proposition* is in Theorems, it becomes the Current Theorem; otherwise, this designation is applied to its parent theorem. If *proposition* is new, it is added to Theorems. *proposition* is normalized and printed. This command is used for

- random access in a proof tree; and

- starting or resuming a proof (but see the description of the resume command RESUMECOMMAND).

type *typeName;*

Specifies *typeName* as the name of an abstract type, whose specification will be given by subsequent commands. The name *typeName* is added to the TypeSet and is pushed onto ContextStack. If *typeName* is already a member of the TypeSet, its existing specification *will be discarded.* Each new type is automatically provided with one variable declaration (the name of which is controlled by the profile entry *DummyVarName*), a declaration of an equality operation, and an axiom explicitly stating that the equality operation is reflexive. The remaining properties of an equality operation are *assumed*, and should be validated by the creator of the type.

undo *[eventNumber];*

Undoes the effects of execution of the command at event *eventNumber*, if possible.

up *[integer];*

Moves the cursor up to a predecessor in the tree. If the Current Proposition is already a theorem, this command has no effect. The number of ascensions defaults to 1.

use *[nodeName,] proposition;*

This command is exactly like the apply command, but also prints the new Current Proposition.

IV.2. Interlisp Commands: Useful Interpreter Commands

DA Prints the time of day. This command is also an operating system executive command.

EXEC Invokes the operating system executive as a subroutine. The user should type POP to return to Interlisp.

IV.3. Interlisp Commands: Useful Editor Commands

These commands can be used only in the Interlisp editor as subcommands of the @ command INTERLISPEDITOR, ATCOMMAND.

!0 Modifies the focus of attention to be the parent of the current expression.

^ Resets the focus of attention to the entire initial expression.

^ 5 2 Modifies the focus of attention to be the sequence of hypotheses of Current Proposition.

hyp Same as ^ 5 2.

con Same as ^ 5 3.

^ 5 3 Modifies the focus of attention to the conclusions of Current Proposition.

n *n* is a positive integer. This command moves the focus of attention to the n^{th} element of the current expression. Caution: the command (*n*) deletes the n^{th} element.

BK Modifies the current expression to be the previous sibling if possible.

(delete *n*)
The n^{th} element of the current expression is deleted.

(delete n_1 n_2 n_3)
The children at the listed positions are deleted. These indices are instantaneous, not one-at-a-time.

eval The current expression is evaluated.

(extract *n*)

The current expression is replaced with its n^{th} child. For example, if the current expression is `(AND e1 e2 e3)` then

The command:	will result in:
(delete 2)	`(AND e2 e3)`
(delete 2 3)	`(AND e3)`
(extract 2)	`e1`

It is not sound to delete operators.

F *pattern*

The F command attempts to find *pattern* within the current expression. If this search is successful then the focus of attention becomes the expression that matches *pattern*. *Pattern* can be any atom, and can contain escapes (which the operating system indicates as $). Each escape can match zero or more contiguous characters in an atom, e.g., VER$ matches VERYLONGATOM. The command will print a message if it cannot find the pattern.

infix The current expression is printed in infix form.

(invoke *definedName*)

The first instance of the definition with name *definedName* in the current expression is expanded.

NX This command moves the focus of attention to the next sibling. For example, if the expression being edited is

(PLUS (FOO 2) (FUM 3))

and the current expression is

(FOO 2)

then the NX command would focus upon

(FUM 3)

This command is very useful after the user uses the *n* command and then discovers that he or she mis-counted.

ok The user is returned to the ***Affirm*** executive, and the modified expression becomes Current Proposition.

Pa This command prints the current expression, showing the structure, (but not the contents) of contained subexpressions, a few levels deep.

PPa This command pretty-prints the current expression.

stop The edit is aborted; no changes are made to Current Proposition, and the user is returned to the ***Affirm*** executive.

References

Index