IN-61
198 pages

JPL PUBLICATION 85-89

IN-14287

# STAR (Simple Tool for Automated Reasoning)

## Tutorial Guide and Reference Manual

Gary C. Borchardt

(NASA-CR-177157) STAR (SIMPLE TOOL FOR AUTOMATED REASONING): TUTORIAL GUIDE AND REFERENCE MANUAL (Jet Propulsion Lab.) 198 p HC A09/MF A01 CSCL 09B

N86-28635

Unclas
G3/61 43404

December 15, 1985

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL PUBLICATION 85-89

# STAR (Simple Tool for Automated Reasoning)

## Tutorial Guide and Reference Manual

Gary C. Borchardt

December 15, 1985

NASA
National Aeronautics and
Space Administration

**Jet Propulsion Laboratory**
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

## Abstract

STAR is an interactive, interpreted programming language for the development and operation of Artificial Intelligence application systems. The language is intended for use primarily in the development of software application systems which rely on a combination of symbolic processing, central to the vast majority of AI algorithms, with routines and data structures defined in compiled languages such as C, FORTRAN and PASCAL. References to routines and data structures defined in compiled languages are intermixed with symbolic structures in STAR, resulting in a hybrid operating environment in which symbolic and non-symbolic processing and organization of data may interact to a high degree within the execution of particular application systems.

The STAR language was developed in the course of a project involving AI techniques in the interpretation of imaging spectrometer data and is derived in part from a previous language called CLIP. The interpreter for STAR is implemented as a program defined in the language C and has been made available for distribution in source code form through NASA's Computer Software Management and Information Center (COSMIC). Contained within this report are the STAR Tutorial Guide, which introduces the language in a step-by-step manner, and the STAR Reference Manual, which provides a detailed summary of the features of STAR.

## Foreword

This report describes STAR, a computer language for the development and operation of Artificial Intelligence application systems. STAR is similar in some ways to the language LISP, but as well contains features and a syntax peculiar to itself, based in part on a previous language, CLIP, developed at the University of Kansas in 1980-81.

In keeping with the structure of AI application systems, a major emphasis in STAR lies in the integration of symbolic processing with numerical or otherwise non-symbolic processing performed in languages such as C, FORTRAN and PASCAL. To this end, STAR includes features which allow it both to manage the execution of external, compiled routines, and, concurrently, to organize and manage the use of various data structures operated upon by these routines. Additional features allow external routines to access and perform operations on the symbolic structures of STAR as well as call arbitrary functions within STAR. In this manner, it is possible to set up a hybrid operating environment in which a single, shared information base containing symbolic and non-symbolic data structures is operated upon alternatively by functions in STAR and in various compiled languages.

The STAR interpreter is implemented as a program defined in the language C and is intended primarily for use on general-purpose computing hardware in combination with the compiled languages resident to a particular system. Source code for the STAR interpreter is available through NASA's Computer Software Management and Information Center (COSMIC), 112 Barrow Hall, University of Georgia, Athens, GA 30602; telephone 1-404-542-3265. The program identification number assigned to STAR is NPO-16832.

Programming in STAR is similar in some ways to programming in other symbolic processing languages such as LISP and PROLOG. Included within the language are seven primitive data types and associated operations for the manipulation of these structures. Symbolic information may be stored in the form of a semantic network with the built-in capability for inheritance of values and generation of side-effects, and programming may be accomplished either through the definition of symbolic functions and associated variables or through the definition of sets of production rules. STAR is an extensible language in the sense that new built-in operations on symbolic structures may be created by defining external, compiled routines which perform these operations. As STAR provides a fairly low-level, general symbolic processing environment, it is intended for use primarily by application programmers already familiar with the variety in AI algorithms and representation schemes.

The STAR Tutorial Guide provides an introduction to the language in a step-by-step manner, including examples and strategies in the use of STAR. In contrast, the STAR Reference Manual describes specific features of the language in isolation, organized categorically for reference while using STAR. Additional information related to STAR may be found in the following two references. The former describes the organization of STAR and its use in the SPECTRUM knowledge-based system for interpretation of imaging spectrometer data. The latter describes the CLIP language which preceded STAR.

Borchardt, G.C., "STAR: A Computer Language for Hybrid AI Applications." In Kowalik, J. (ed.), *Coupling Symbolic and Numerical Computing in Expert Systems,* North-Holland, Amsterdam, 1986.

Borchardt, G., and Bavel, Z., *CLIP: Computer Language for Idea Processing,* TR-81-4, Department of Computer Science, University of Kansas, Lawrence, KS, 1981.

The author wishes to thank Professor Zamir Bavel of the University of Kansas for helpful guidance and criticism in the development of the original CLIP language.

## Table of Contents

STAR Reference Manual (continued)

# A Tutorial Guide to STAR

## Part I: STAR Data Types

### 1. Using STAR

There are two distinct levels in the development of an application system using STAR. At the first level, routines and data structures may be defined separately in general-purpose compiled languages, to be linked together with the STAR interpreter as a single executable module on the host computer. At the second level, routines, rules and other symbolic structures are defined within STAR, or their definitions are loaded in symbolically from a text file constructed through the use of a general-purpose text editor. At this level, STAR is used in an interactive manner similar to the use of languages such as LISP or APL. Segments of the application system may be tested out in isolation, and run-time debugging and modification of these segments may be conducted.

When parts of an application system appear to be in fairly stable operational form, the STAR definitions applying to these parts may be loaded in and the entire STAR interpreter suspended in its enhanced state, producing a new version of the STAR interpreter which contains these definitions initially and which may be used for subsequent development sessions in place of the original version of STAR. A complete application system as produced by the suspending operation contains the STAR interpreter, possibly linked together with routines and data structures defined in general-purpose languages, and further enhanced with symbolic definitions inside STAR. The assembled package may be called directly from the operating system with directions to begin execution with a call to a specified routine. The user of the application system is thereby taken directly into the application environment and is shielded from the operation of STAR.

Parts I, II and III of the Tutorial Guide concentrate on the symbolic processing environment of STAR, apart from the added aspects of linking STAR with external routines and data structures. Following this, Part IV addresses the issues related to external definitions and the overall process of compiling, linking, loading symbolic definitions and suspending the enhanced state of the STAR interpreter. Section 19, at the end of Part IV, provides an overview of the use of STAR in the development of SPECTRUM, a knowledge-based system developed at JPL for the geological interpretation of imaging spectrometer data.

### 2. UNITs and Evaluation

The syntax and operation of the STAR language centers around a set of seven data types called UNITs. The individual types are: NUMBERs, TOKENs, STRINGs, LISTs, RECORDs, EXPRESSIONs and CONNECTIONs; examples are given below. All data structures, routines, production rules and other programming structures in STAR are constructed using UNITs.

NUMBERs: 23.4, -500, 0.724e4,

TOKENs: GOAL, MULTIPLY,

STRINGs: "2:55:52 PDT",

LISTs: [88 79 92 86 89],

RECORDs:
```
{name -> BINARY_SEARCH
 member_of -> function
 arguments -> [tree test]
 temporary -> [ii]
 algorithm ->
     [if(leaf(.tree) 'return(.tree))
      set(ii apply(.test [.tree]))
      if( =(.ii 0) 'return(.tree))
      return(
       binary_search(
        ifelse( >(.ii 0)
         'left(.tree)
         'right(.tree)
         )
        .test))
     ]
},
```

EXPRESSIONs: +(12 43), binary_search(tree_1 test_1),

CONNECTIONs: ^IMAGE_FILTER_FUNCTION, ^ARRAY_373220.

As a quick summary, NUMBERs, TOKENs and STRINGs are used to represent individual data values, LISTs and RECORDs are used to represent structured data values, EXPRESSIONs are used to represent instructions to the STAR interpreter, and CONNECTIONs are used to reference routines and data structures defined elsewhere in general-purpose computer languages. RECORDs with "name" fields (the RECORD above, for example) are used extensively in STAR and are referenced in a special manner. All lowercase words in the above examples are instances of this referencing device and refer to particular named RECORDs. (This referencing device is explained in detail in Part II of the Tutorial Guide.)

In the normal interactive mode of the STAR interpreter, the programmer enters a UNIT to be evaluated, the interpreter displays the result (also a UNIT), and the process repeats until the programmer signals an end to the session. The UNIT entered by the programmer is almost always an EXPRESSION, as EXPRESSIONs alone specify commands for action on the part of the STAR interpreter. An example of an EXPRESSION entered for evaluation is given below, along with the UNIT (in this case, a LIST) resulting from evaluation. Here, the programmer has invoked the function "intersection," causing a

set intersection operation to be performed on the two LISTs given as arguments.

**Example 1:**

```
intersection(
 [2 4 5 3 1 7]
 [8 6 3 2]
 )

[2 3]
```

Note the indentation of the UNIT entered by the programmer. UNITs entered for evaluation are automatically buffered by a five space margin. UNITs returned as results are listed starting in column zero.

Evaluation performed at the level of interaction with the programmer is basically the same as evaluation of UNITs anywhere within STAR, but there are a few differences. First of all, when the programmer enters a UNIT to be evaluated, the interpreter parses the UNIT for syntactic form directly as it is entered. This allows the interpreter to (1) warn immediately of syntax errors, displaying the offending character and allowing the programmer to continue from the point prior to that character, and (2) automatically indent the UNIT as it is entered, five spaces plus one space for each level of nesting (Example 1 involves a single level of added indentation for the last three lines of the EXPRESSION).

In addition, STAR recognizes four printable characters and two non-printable characters as "escapes," as summarized below.

```
   $  - in-line comment specification,
   @  - "preprocessing" of a UNIT,
   %  - backtracking to previous nesting levels,
   #  - recalling the last UNIT returned by STAR,

delete  - deleting last character entered,
ctrl C  - interrupting evaluation of present UNIT.
```

These characters are extremely useful in smoothing the interactive cycle with the STAR interpreter and are thus described in some detail in this section, prior to the introduction of most of the major aspects of STAR. The following example illustrates the use of the four printable escape characters. These characters may be used at any syntactic division in the construction of a UNIT.

**Example 2:**

```
append(              $ Concatenation of two LISTs.
 [1 2 3 4 5]
 [7 8 9%             $ <--- Back up one level.
 [6 7 8 9 0]
 )

[1 2 3 4 5 6 7 8 9 0]

intersection(        $ Intersection of the above
 #                   $   LIST with a second LIST.
 [11 3 6 @*(2 2)]    $ <--- Preprocessing of a UNIT.
 )

[3 4 6]
```

When a dollar sign ("$") is entered by the programmer, all characters including the dollar sign and following it up to the end of the input line are ignored. This is one way of documenting source files or transcripts of sessions. (Another mechanism for commenting uses STRINGs, as will be seen later.)

The percent sign ("%") allows for recovery from mistakes by causing the STAR interpreter to backtrack to the point just preceding the beginning of its current level of nesting. Thus, in the above example, the percent sign entered causes a backtracking to the point just prior to the start of the LIST "[7 8 9...". Multiple percent signs may be used to backtrack more than one level.

The pound sign ("#") retrieves the last UNIT displayed by STAR. In the case of the above example, this is the LIST "[1 2 3 4 5 6 7 8 9 0]".

Finally, the "at" sign ("@") causes the preprocessing of a UNIT, the result of which is placed in the entered UNIT before normal processing. In the above example, the EXPRESSION "*(2 2)" is evaluated, producing the NUMBER "4" which is placed in the LIST. Normally, the evaluation of a LIST such as the above protects the elements themselves from evaluation.

The two non-printable escape characters are used as follows. The "delete" character causes a backtracking of the cursor on a video terminal, allowing the programmer to make single character corrections. The "control C" character causes the STAR interpreter to cease an ongoing parse, evaluation or display of a UNIT and return to the top level, ready to accept the next UNIT. This character is indispensable in recovering from endless loops or mistaken lengthy calculations.

In addition to the above, two remaining aspects of STAR require introduction at this point before proceeding: exiting the STAR interpreter and the generation of error messages. An exit from the STAR interpreter is generated by a call to the function "exit" as follows.

exit()

The "exit" function takes no arguments.

Error messages are generated in STAR whenever the evaluation of an EXPRESSION cannot be carried out for some reason or other. These messages fall into six categories as listed below.

**DEFINITION ERROR:** The function invoked by the EXPRESSION or an integrally-related structure is improperly defined.

**APPLICATION ERROR:** An improper number of arguments is specified for the function invoked by the EXPRESSION.

**TYPE ERROR:** The indicated argument is of an incorrect UNIT type (NUMBER, TOKEN, STRING, LIST, RECORD, EXPRESSION or CONNECTION).

**CLASS ERROR:** The indicated argument is a named RECORD belonging to the wrong class. (The concept of named RECORDs and classes is described in Section 8.)

**VALUE ERROR:** The indicated argument is of the correct type (and class, if applicable), but the particular value is out of range or otherwise unusable.

**PERMISSION ERROR:** The indicated argument must not be altered in the manner specified, as this could cause inconsistencies or unpredictable effects in the operation of STAR.

Whenever an error is detected in STAR, a message is generated at the user terminal specifying the type of the error, the offending UNIT, and providing a trace listing of all pending functions and/or structures. The following notations appear in the trace listing.

| | |
|---|---|
| xxx(n | : error detected prior to application of xxx, while preparing its n'th argument, |
| xxx(n* | : error detected during application of xxx, relating to its n'th argument, |
| [n | : error detected while processing the n'th element of a LIST, |
| {aaa | : error detected while processing the value of attribute aaa in a RECORD, |
| .....xxx( | : error detected while applying the user-defined or external function xxx. |

The generation of error messages is covered in greater detail in the STAR Reference Manual.

## 3. NUMBERs and STRINGs

### NUMBERs

A NUMBER in STAR is a sequence of digits, possibly preceded by a minus sign ("-") and followed by an optional decimal point ("."), an optional set of digits describing the fractional component, and an optional exponent specification. The exponent specification has the form "e" or "E" followed by an optional "+" or "-" and one or more digits. All NUMBERs are considered to be of the floating point variety; however, NUMBERs with no fractional components are displayed as integers. The following are all NUMBERs:

```
23 , 004 , -3.45 , 1002.9997 , 340.34e-5.
```

STAR contains several built-in functions for use in EXPRESSIONs involving NUMBERs. Five basic operations on NUMBERs are accomplished by the functions "negate", "add", "subtract", "multiply" and "divide". As is the case with a number of other built-in functions in STAR, these functions may be specified in EXPRESSIONs in a "standard" form, and, in addition, in an "abbreviated" form. The two forms may be used interchangeably for functions with designated abbreviations; all other functions may be specified only in the standard form. Examples involving the use of these functions are given below.

**Example 3:**

```
    add(2 multiply(3 5.2))    $ Standard form.

17.6

    +(2 *(3 5.2))             $ Abbreviated form.

17.6

    subtract(
     divide(3 2)
     negate(-44)
     )

-42.5
```

```
-(/(3 2) --44)          $ Abbreviations for
                        $  subtract, negate
-42.5                   $  and divide.
```

There are actually two varieties of the abbreviated form for EXPRESSIONs. One involves the substitution of a predesignated symbol for the name of a function; this is the case for the functions "add", "subtract", "multiply" and "divide" (using the symbols "+", "-", "*" and "/", respectively). The second variety is allowed only for particular built-in functions taking a single argument. This variety involves replacing both the function name and the parentheses enclosing the argument with a single prefix symbol, as for the function "negate", above.

It may be noted that the hyphen character has three separate uses. These uses are strictly partitioned as follows. When followed by a digit, a hyphen specifies the start of a negative NUMBER. When followed by an EXPRESSION or a second hyphen, it specifies the abbreviated form of the "negate" function (thus, "--44", above, is actually an EXPRESSION applying the "negate" function to the NUMBER "-44"). Finally, when followed by a left parenthesis, it specifies the abbreviated form of the "subtract" function. The "negate" and "subtract" functions are distinguishable because "negate" may be abbreviated only in the non-parentheses form, while "subtract" may be abbreviated only in the included-parentheses form.

Two other built-in functions exist for the manipulation of NUMBERs: "minimum" and "maximum". These functions each take two NUMBERs as arguments and return a NUMBER as their result. As well, other functions exist which use NUMBERs in some way or another; these functions will be introduced in the context of their application.

## STRINGs

A STRING is a sequence of characters, possibly containing one or more carriage returns, and enclosed in double quotes. STRINGs which do contain carriage returns are prefixed with an additional double quote at the beginning of each line of text. This is a necessary mechanism for avoiding confusion between the indentation of UNITs and spaces inserted within multiple-line STRINGs. A few examples will make this clear. The following are all syntactically acceptable STRINGs.

```
"A single line STRING."

"A multiple line STRING,
"containing two carriage
"return characters."
```

```
[
 ["A multiple line STRING
  "nested within two levels
  "of LISTs."
  ]]
```

When a STRING is entered by the programmer, the STAR interpreter automatically provides a preliminary double quote character for each line after the first. This action on the part of STAR works in conjunction with the automatic indentation provided by STAR: the preliminary double quote is essentially the indentation for the pending STRING. The completion of a STRING is indicated by the programmer through the entering of a final double quote character following all characters of text.

As is common practice in a number of programming languages, a double quote character may be entered within a STRING by typing in two double quote characters. In conjunction with the use of lefthand double quotes for succeeding lines of text, however, this may cause some confusion on the part of the programmer. It is useful to note that no syntactic ambiguity exists, regardless of the combination, and it is always possible to determine which double quote characters delimit the boundaries of the STRING and which double quote characters specify the containment of double quotes within the STRING.

STAR contains six built-in functions for the direct manipulation of STRINGs. These functions are illustrated in the example below.

**Example 4:**

```
    character("abcdefgh" 4)        $ Return the fourth
                                   $   character as a
"d"                                $   STRING.

    character("abcdefgh" -3)       $ Return the third char-
                                   $   acter from the end.
"f"

    fetch("abcdefgh" 5)            $ Return a STRING con-
                                   $   taining the first 5
"abcde"                            $   characters.

    fetch("abcdefgh" -5)           $ ... containing the
                                   $   last 5 characters.
"defgh"

    release("abcdefgh" 1)          $ Return a STRING con-
                                   $   taining all but the
"bcdefgh"                          $   first character.
```

```
    join(                          $ Concatenate 2 STRINGs.
     "First line,                  $   <-- start first STRING.
     ""                            $   <-- end first STRING.
     "Second line."                $   <-- second STRING.
     )

"First line,
"Second line."

    find("the" "move the box")     $ Number of characters
                                   $   to be "released" to
5                                  $   reach the start of
                                   $   the sub-STRING.

    find("a" "move the box")       $ If sub-STRING is not
                                   $   found, return length
12                                 $   of second STRING.

    length("move the box")         $ Number of characters.

12
```

A number of built-in functions in STAR make use of the sign for a numerical argument in determining a direction of processing. Such is the case with the STRING functions "character", "fetch" and "release". Each of these functions takes a positive NUMBER to indicate counting from the head of the STRING and a negative NUMBER to indicate counting from the tail of the STRING. The NUMBER "0" is not allowed for the function "character", causes an empty STRING to be returned for "fetch" and results in a duplicate copy of the original STRING for "release".

STRINGs in STAR are the catch-all data type, as they can be used to represent items of varying syntactic construction. In addition, any UNIT can be converted into a STRING of its displayed form through the use of the "spell" function, and a STRING describing a UNIT can be parsed and converted into that UNIT through the use of the "unspell" function. A few simple examples involving the "spell" and "unspell" functions are given below.

**Example 5:**

```
spell(                                  $ Convert to a STRING.
 {name -> SQRT
  member_of -> function
  comment ->
      "     (NUMBER) => NUMBER
      "
      "     Square root of a NUMBER."
  n_arguments -> 1
  algorithm -> ^EXTERNAL_SQRT_FUNCTION
  })

"{name -> SQRT
" member_of -> function
" comment ->
"     ""     (NUMBER) => NUMBER
"     ""
"     ""     Square root of a NUMBER.""
" n_arguments -> 1
" algorithm -> ^EXTERNAL_SQRT_FUNCTION
" }"

   unspell(#)                           $ Using the above STRING
                                        $   as an argument.
{name -> SQRT
 member_of -> function
 comment ->
     "     (NUMBER) => NUMBER
     "
     "     Square root of a NUMBER."
 n_arguments -> 1
 algorithm -> ^EXTERNAL_SQRT_FUNCTION
 }
```

Note that the double quote characters in the STRING returned by "spell" are displayed in pairs, two quotes for each actual quote in the STRING. This is the convention in STAR; each double quote contained within a STRING is always displayed as a pair of double quotes. By this device, a STRING is always displayed exactly as it would be entered on the terminal, and thus it is possible to save the displayed version of a STRING on file and read it in at a later time with no ill side effects.

An additional function, "scan", is introduced in Section 12 and parallels the operation of "unspell". Whereas syntax errors are announced when detected in the operation of "unspell", the "scan" function is silent in such cases, merely returning a predesignated flag value as its result.

### Evaluation of NUMBERs and STRINGs

NUMBERs and STRINGs are both unaffected by evaluation. This is true whether they are entered singly by the programmer (in which case they are simply echoed) or nested within other structures which are to be evaluated.

## 4. LISTs and RECORDs

The LIST and RECORD types in STAR are the major components in the construction of complex data structures. These two types balance each other in the sense that LISTs are geared towards the combination of like elements and RECORDs are geared towards the combination of unlike elements. Both structures may contain an arbitrary number of entries, with the entries in RECORDs always occurring as attribute-value pairs of UNITs.

### LISTs

The LIST type in STAR is somewhat different from the list structure of LISP or PROLOG in that it is not defined in terms of a lower-level structure such as the "dotted-pair". A LIST in STAR is simply an ordered sequence of zero or more UNITs. Syntactically, it amounts to a pair of brackets ("[" and "]") surrounding zero or more UNITs which are separated by spaces or carriage returns.

STAR contains seven built-in functions for the low-level processing of LISTs and three built-in functions for the processing and use of LISTs as sets. These functions are illustrated in the following example.

### Example 6:

```
    select([1 3 5 7 9] 2)          $ Return the second
                                   $   element.
3

    select([1 3 5 7 9] -1)         $ Return the first element
                                   $   counting from the end
9                                  $   of the LIST.

    replace([A B C] 2 ZZZ)         $ Return a copy of the
                                   $   LIST replacing the old
[A ZZZ C]                          $   second element.

    replace([A B C] -3 ZZZ)        $ ...replacing the third
                                   $   element from the end.
[ZZZ B C]
```

```
    delete(["*" "+" ":"] -3)        $ Return a copy of the
                                    $   LIST minus the third
["+" ":"]                           $   element from the end.

    insert([1 2 3] 2 [])            $ Return a copy of the
                                    $   LIST with the new UNIT
[1 [] 2 3]                          $   in the 2nd position.

    insert([1 2 3] -1 [])           $ ...with the new UNIT in
                                    $   the first position
[1 2 3 []]                          $   counting from the end.

    take([1 3 5 7 9] 4)             $ Return a LIST containing
                                    $   the first 4 elements
[1 3 5 7]                           $   of the original LIST.

    drop([1 3 5 7 9] -4)            $ Return a LIST containing
                                    $   all but the last four
[1]                                 $   elements.

    append(["*"] ["+"])             $ Concatenate two LISTs.

["*" "+"]

    size([A B C D E])               $ Number of elements in a
                                    $   LIST.
5

    union([1 2 4] [1 3 5])          $ Set union. (See descrip-
                                    $   tion for the built-in
[1 2 4 3 5]                         $   function "equal").

    intersection(                   $ Set intersection.
     [1 4 2 6 3 8 0 9]
     [3 7 5 4 1 2]
     )

[1 4 2 3]

    difference(                     $ Set difference.
     [3 7 5 4 1 2]
     [1 4 2 3]
     )

[7 5]
```

There are some obvious parallels between the LIST functions described above and the STRING functions presented previously. The functions "character" and "select", "fetch" and "take", "release" and "drop", "join" and "append", and "length" and "size" all behave in predictably similar manners, with the first mentioned of each pair operating on STRINGs and the second mentioned operating on LISTs.

The functions "select", "replace", "delete", "insert", "take" and "drop" are also of the variety which makes use of the sign of its numerical argument: a positive NUMBER indicates counting from the start of a LIST, and a negative NUMBER indicates counting from the end of the LIST.

The function "insert" is slightly different from its associated functions "select", "replace" and "delete" in that the numerical argument is allowed to be one greater in magnitude than the current number of elements in the LIST. In all cases, the numerical argument to "insert" specifies the final position to be assumed in the LIST by the UNIT to be inserted. Other elements may be repositioned with respect to one or the other end of the LIST as a result.

One final note should be made concerning the LIST manipulation functions. As with the built-in functions concerning NUMBERs and STRINGs, all of the functions listed above are non-destructive regarding their arguments. In cases where a LIST is returned as a result, this is derived from a copy of the LIST supplied as an argument.

## RECORDs

RECORDs are similar to LISTs in that they are structurally uniform and are not composed of separable parts other than the UNITs contained within them plus the outer "shell" of brackets or braces. A RECORD consists of a pair of braces ("{" and "}") surrounding zero or more pairs of UNITs, with the pairs themselves delimited by spaces or carriage returns and the individuals within each pair delimited by the two-character symbol of hyphen-greater-than ("->"), possibly padded with spaces or carriage returns on either side of the symbol. The following are all syntactically acceptable RECORDs.

```
{height->5 width->6 length->2}

{time -> 14  value -> true}

{name -> A
 member_of -> variable
 bindings -> [4 5]
 }
```

Within a pair of UNITs forming an entry in a RECORD, the first UNIT is referred to as the "attribute" and the second as its "value". The attribute is itself a RECORD of the "named" variety. (This is described more fully later; for the purpose of this discussion, these may be considered simply as lowercase words.) STAR contains an initial set of built-in attributes for use in programming, and further attributes may be defined by the programmer for use in a given application system (see Example 7, below). The value

associated with an attribute in a RECORD may be any type of UNIT.

A broad distinction is made in STAR between RECORDs which contain an entry for the attribute "name" and those which do not. RECORDs containing a "name" entry are given a special status allowing them to be stored as nodes in STAR's semantic network, referenced directly using the "lowercase word" device and destructively (directly) manipulated by built-in STAR functions. The mechanisms associated with named RECORDs are deferred until later in the tutorial.

Apart from this distinction, however, RECORDs may be used simply as generalized data structures similar to LISTs. A set of built-in functions exists in STAR for the non-destructive manipulation of RECORDs of all types (that is, a copy of the RECORD is modified in each case, leaving the original intact). These functions are illustrated below. STRINGs are used as values in the RECORDs below for simplicity. In the general case, UNITs of all types are used in this context.

**Example 7:**

```
    create(
     type
     attribute
     )

{name -> TYPE
 member_of -> attribute
 }

    create(
     color
     attribute
     )

{name -> COLOR
 member_of -> attribute
 }

    create(
     status
     attribute
     )

{name -> STATUS
 member_of -> attribute
 }
```

```
$ These definitions are
$  explained more fully
$  in later sections.
$
$ Particular attributes
$  used as lefthand
$  members in RECORD
$  entries must each be
$  created in the
$  manner shown.
$
$ Following the creation
$  of such definitions,
$  individual attributes
$  may appear within
$  RECORDs throughout
$  STAR by their lower-
$  case reference names
$  ("type", "color" and
$  "status", here). The
$  attributes "name" and
$  "member_of", along
$  with several other
$  attributes, are
$  predefined in STAR.
```

```
    get(                                $ Return the value of
     {type -> "SQUARE"                  $   the attribute "color"
      color -> "RED"}                   $   in the given RECORD.
     color
     )

"RED"

    put(                                $ Return a copy of the
     {type -> "SQUARE"                  $   RECORD with a new
      color -> "RED"}                   $   value for "color".
     color
     "GREEN"
     )

{type->"SQUARE" color->"GREEN"}

    put(                                $ Alternate use of "put"
     {type -> "CIRCLE"}                 $   to create a new entry
     color                              $   in the returned  copy
     "YELLOW"                           $   of the RECORD.
     )

{type->"CIRCLE" color->"YELLOW"}

    omit(                               $ Return a copy of the
     {type -> "BOX"                     $   RECORD which omits an
      status -> "FULL"}                 $   entry for "status".
     status
     )

{type->"BOX"}

    detach(                             $ Return a copy of the
     {type -> "BUCKET"                  $   RECORD with only a
      color -> "BLUE"                   $   subset of its
      status -> "FULL"}                 $   entries, ordered as
     [color type]                       $   specified in the
     )                                  $   LIST.

{color->"BLUE" type->"BUCKET"}
```

```
    attach(                          $ Return a copy of the
     {type -> "CIRCLE"}              $   first RECORD,
     {type -> "PAINT"                $   including attributes
      color -> "GREEN"               $   which appear in the
      status -> "WET"                $   second but not the
      }                              $   first, along with
     )                               $   values specified in
                                     $   the second RECORD.
{type -> "CIRCLE"
 color -> "GREEN"
 status -> "WET"
 }

    key(                             $ Return a LIST of the
     {type -> "DOG"                  $   attributes appearing
      status -> "YOUNG"              $   in a RECORD.
      })

[type status]

    image(                           $ Return a LIST of the
     {type -> "DOG"                  $   values appearing in a
      status -> "YOUNG"              $   RECORD.
      })

["DOG" "YOUNG"]

    build(                           $ Construct a RECORD from
     [type status]                   $   a LIST of attributes
     ["DOG" "YOUNG"]                 $   and a LIST of values.
     )

{type->"DOG" status->"YOUNG"}
```

Again, there are similarities between the built-in functions for the manipulation of RECORDs and built-in functions for the manipulation of LISTs and STRINGs. The parallels are not as direct in this case, however. The functions "get", "put" and "omit" are used to perform low-level access and modification of RECORDs, with the function "put" assuming the dual role of modifying existing entries and adding new entries. The functions "detach" and "attach" are used to selectively remove a portion of a RECORD or to merge two RECORDs, giving priority to entries of the first RECORD. It should be emphasized that in all cases, the original RECORD used as an argument is unaltered by the process, and that the result returned is derived from a copy of that RECORD.

## Evaluation of LISTs and RECORDs

It is important to note that in STAR the evaluation of LISTs and RECORDs leaves these structures unchanged. That is, the elements contained within LISTs and RECORDs are protected from evaluation. This is necessary in the support of the semantic network in STAR, which is composed of an extensively interwoven nesting of LISTs and RECORDs within another. If the elements of LISTs and RECORDs were to be evaluated automatically as their outer structures were evaluated, a condition of infinitely recursive evaluation would present itself with the evaluation of a single structure in the network. To counteract this threat, STAR protects the elements of LISTs and RECORDs from evaluation and provides a built-in function, "prepare", to be used in cases which do require the evaluation of elements in a LIST or RECORD. The use of this function is illustrated below.

**Example 8:**

```
    [+(2 3) [+(2 3)]]          $ Two EXPRESSIONs nested 1 and
                               $ 2 levels deep, respectively.
[+(2 3) [+(2 3)]]              $ No change from evaluation.

    prepare(#)                 $ Forced evaluation of the top
                               $   level of elements.
[5 [+(2 3)]]

    {a -> +(2 3)               $ RECORD containing nested
     b -> [+(2 3)]             $   EXPRESSIONs.
     }

{a->+(2 3) b->[+(2 3)]}

    prepare(#)                 $ Evaluate the top level of
                               $   elements in the above
{a->5 b->[+(2 3)]}             $   RECORD.
```

The argument to the function "prepare" may be a UNIT of any type. The difference between the operation of "prepare" and direct evaluation, however, is only evident in its application to LISTs and RECORDs.

## 5. The Remaining UNITs

The remaining three UNIT types, TOKENs, EXPRESSIONs and CONNECTIONs, are used differently from the general-purpose data types, NUMBERs, STRINGs, LISTs and RECORDs. UNITs of these three types are not manipulated to the extent of the other types and, correspondingly, have fewer built-in functions concerned with their use.

### TOKENs

TOKENs are composed of a single capital letter followed by zero or more occurrences of capital letters, digits and the underscore character ("_"). They are used chiefly as atomic values and usually in the sense of supplying names to RECORDs. TOKENs are not decomposable and thus they are used only in limited contexts requiring "face-value-only" data values.

### EXPRESSIONs

EXPRESSIONs are used primarily to store commands to the STAR interpreter in the form of data. When the commands embodied in EXPRESSIONs are to be invoked, the EXPRESSIONs are simply evaluated. In some cases, however, it is also useful to be able to create, modify and disassemble EXPRESSIONs in the course of operation in STAR. Three built-in functions are provided for this purpose, as described below.

**Example 9:**

```
   operation('*(+(2 1) 3))          $ Return the function
                                    $   being applied in
{name -> MULTIPLY                   $   the EXPRESSION.
 member_of -> function
 comment ->
    "     (NUMBER NUMBER) => NUMBER
    "
    "     Multiplication."
 abbreviation -> "*"
 n_arguments -> 2
 algorithm -> ^C_MULTIPLY_FUNCTION
 }

   application('*(+(2 1) 3))        $ Return a LIST of the
                                    $   unevaluated argu-
[+(2 1) 3]                          $   ments.

   formulate(multiply               $ Construct an EXPRES-
    [+(2 1)                         $   SION from a func-
     3]                             $   tion and a LIST of
    )                               $   arguments.

*(+(2 1) 3)
```

Introduced in the above example is the "quote" function (abbreviated "'"), similar to the quote function of LISP. This function takes one argument, which it spares from evaluation. The quote function is the only function which can circumvent the evaluation of UNITs and is useful in the entering of EXPRESSIONs which are to be retained in the

form of data structures and manipulated by functions such as "operation" and "application". It may be recalled that placement of UNITs within LISTs and RECORDs is another mechanism by which entered UNITs may be spared from evaluation: this is the mechanism which permits the definitions of functions such as "binary_search" in Section 2 to be entered without their contained EXPRESSIONs being evaluated immediately. Because of the "protection" from evaluation of elements in LISTs and RECORDs, the "quote" function is not used as heavily in STAR as in LISP.

Once an EXPRESSION has been entered and spared from evaluation, it may be decomposed through the use of the functions "operation" and "application". The function "formulate" works in the opposite direction, forming an EXPRESSION from a function and a LIST of arguments. Through the use of these functions, direct manipulation of EXPRESSIONs is avoided; in lieu of this, substituted manipulation of functions and LISTs of arguments, coupled with the transformations provided to and from the EXPRESSION type, is used.

## CONNECTIONs

CONNECTIONs in STAR are references to quantities outside of the STAR language. These may be not only routines defined in other programming languages, but data structures as well. Thus, STAR is able to coordinate entire program/data assemblages defined in other languages.

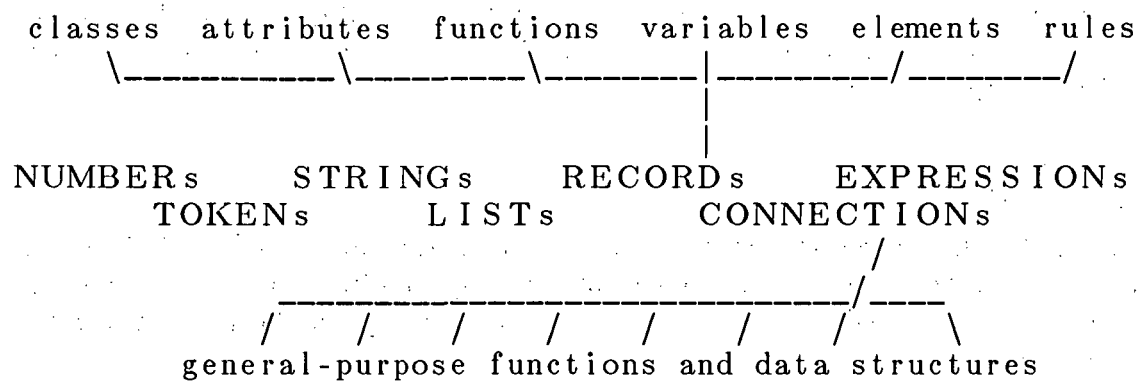
CONNECTIONs are displayed as a circumflex character ("^") followed by a label assigned individually to each CONNECTION. CONNECTIONs which refer to external routines are stored in function definitions very similar to the definitions for the built-in functions of STAR. These routines may then be applied in the same manner as any other STAR function. CONNECTIONs which refer to external data structures are stored within other UNITs according to the design of the application. These CONNECTIONs may then be retrieved later and used as arguments when calling external routines.

The notion of data exchange between STAR and various general-purpose computer languages relies on two components: (1) the ability of STAR UNITs to contain CONNECTIONs which refer to external routines and data structures, and conversely, the ability of general-purpose languages to store pointers to STAR UNITs within their own data structures, and (2) the ability for both STAR and the general-purpose languages to call functions of the other portion, and specifically, the availability of a set of approximately 50 utility functions accompanying the STAR interpreter which may be called by external functions to access, create and modify UNIT structures.

CONNECTIONs, the calling of external functions and external manipulation of UNIT structures are covered in detail in Part IV of the Tutorial Guide. In addition, Sections 2.7, 5 and 6 of the Reference Manual summarize various aspects of STAR related to these features.

## 6. A Hierarchy of Data Types

The data structures defined and used in STAR range considerably in complexity -- from general-purpose data structures defined in other languages and included through the use of CONNECTIONs -- to UNITs of all types, available for general symbolic processing -- and finally to a group of "named RECORDs", used specifically in the context of building high-level structures for AI application systems. These three broad divisions of data types may be considered in the form of a hierarchy, allowing the manipulation of a wide range of data from numerically-oriented data structured for efficiency to symbolically-oriented data structured for flexibility. A diagram of this hierarchy is given below.




The general processing arena in STAR is that of the UNIT, depicted as the middle level above. RECORDs are distinguished in that they provide the framework for a set of higher level constructs, the named RECORDs. The six major divisions of named RECORDs are listed above: "classes," "attributes", "functions", "variables", "elements" and "rules." Individual data structures in each of these divisions are constructed through the use of RECORDs in specialized ways. In the other direction, the UNIT type CONNECTION provides a link to a set of lower-level constructs, the routines and data structures defined in other languages and linked together with STAR for a particular application. In the design of a complete AI system, it is useful to consider these three levels of data within STAR. Representation of particular data quantities in structures at the correct level can be a critical factor in efficiency and flexibility of the application system as a whole.

The next few sections describe the top level of this hierarchy, the various types of named RECORDs and their use in STAR. Following this, the lower level of the hierarchy is discussed and a sample application involving all three levels is worked out in some detail.

## Part II: The Semantic Network

### 7. Referencing Named RECORDs

Several times in the above sections, mention has been made of a distinction between "named" and "unnamed" RECORDs in STAR. RECORDs which contain a value for the attribute "name" are given special treatment and form a higher level of structures, whereas RECORDs which do not contain a value for "name" are treated in a simpler fashion similar to LISTs and other UNIT types. Several examples of each type of RECORD have appeared in the above sections.

One major distinction which sets named RECORDs apart from unnamed RECORDs is that any named RECORD stored within the system may be accessed directly by typing in a lowercase form of the TOKEN listed as its name. This lowercase form is referred to as a RECORD's "reference name". Specifically, all uppercase letters in the "name" TOKEN are converted to lowercase, leaving digits and the underscore character ("_") as they are. Thus the RECORD named with the TOKEN "ADD" has a reference name of "add", the RECORD named with the TOKEN "BOX_2A" has a reference name of "box_2a", and so forth. In order that reference names correspond to unique named RECORDs in STAR, a restriction is made that only one RECORD having a particular TOKEN as its name may be known to the STAR interpreter at any given time.

Named RECORDs are automatically retrieved by STAR when their reference names are entered at the keyboard. For example, the RECORD defining the function "select" in the STAR knowledge base,

```
{name -> SELECT
 member_of -> function
 comment ->
     "     (LIST1 NUMBER1) => UNIT
     "
     "     Return the NUMBER1'th element of LIST1.
     "If NUMBER1 < 0, select the |NUMBER1|'th element
     "from the end of LIST1."
 n_arguments -> 2
 algorithm -> ^C_SELECT_FUNCTION
 },
```

may be accessed directly by typing in its reference name "select", followed by a carriage return. A related activity occurs when the programmer types in an EXPRESSION such as

```
select([10 20 30 40] 2).
```

When the input text for this EXPRESSION is parsed by the STAR interpreter, the RECORD defining the function "select" is located and explicitly referenced as a

component in the internal representation for the entered EXPRESSION.

When a reference name is entered for which there is no existing named RECORD in the knowledge base, STAR creates such a RECORD, and all subsequent uses of that reference name refer to that particular named RECORD. In this manner, STAR can handle "forward references", as, for example, when an EXPRESSION involving one function is entered in the definition of another function, but the first function itself is not defined until a later time, still prior to its first usage in the second function.

It will be noticed that the RECORD defining the function "select", given above, refers to a number of other named RECORDs -- "name", "member_of", "function", "comment", "n_arguments" and "algorithm". These in turn refer to other named RECORDs and so on, forming a highly interwoven nesting of named RECORDs contained one within another. The entire interconnected network comprises the STAR knowledge base. While the named RECORDs in the STAR knowledge base are each accessible in a number of ways from other named RECORDs in the knowledge base, the network itself is self contained and isolated from the context of entering UNITs at the keyboard. The primary mechanism by which the programmer can access named RECORDs in the knowledge base is by entering their reference names at the keyboard.

Due to the highly interwoven nature of the STAR knowledge base, named RECORDs are also usually displayed using their reference names. A named RECORD, when displayed not at the outermost level but *within* a UNIT of any type, is always indicated by reference name only (see the RECORD for "select", above).

One useful side-effect of the reference name mechanism is that it is possible to "poke around" or explore various items in the STAR knowledge base quite easily. One simply enters the reference name for a desired RECORD (e.g., "select"), and the stored RECORD in its expanded form is displayed by the STAR interpreter. Within that RECORD, other named RECORDs are referenced, each of which may be viewed separately by typing in its reference name.

Accessing named RECORDs is accomplished primarily through the use of reference names, but in addition there are two built-in functions in STAR which aid in this task. The first function is "locate", which takes a TOKEN as an argument and returns the corresponding RECORD having that TOKEN as its name, creating a new named RECORD if necessary. The second function is "test", which operates in the same manner, except that it does not create a new RECORD if one does not exist; rather, it simply returns a predesignated flag value of "nil". (The named RECORD "nil" is used often in STAR to represent the absence of some other value. Other "flag value" RECORDs are the boolean values "true" and "false".) The use of the functions "locate" and "test" is illustrated in the following example.

**Example 10:**

```
add                    $ Entering a reference name to
                       $   access a named RECORD.
```

```
{name -> ADD
 member_of -> function
 comment ->
     "     (NUMBER1 NUMBER2) => NUMBER
     "
     "     Addition."
 abbreviation -> "+"
 n_arguments -> 2
 algorithm -> ^C_ADD_FUNCTION
 }

    locate(CHARACTER)        $ Use of "locate" to reference
                             $   an existing named RECORD.
{name -> CHARACTER
 member_of -> function
 comment ->
     "     (STRING1 NUMBER1) => STRING
     "
     "     Return a single-character STRING containing
     "the NUMBER1'th character of STRING1.  If NUMBER1
     "< 0, count from the end of STRING1."
 n_arguments -> 2
 algorithm -> ^C_CHARACTER_FUNCTION
 }

    locate(COSINE)           $ Use of "locate" to create a
                             $   new named RECORD.
{name -> COSINE
 }

    test(CHARACTER)          $ Use of "test" with an existing
                             $   named RECORD.
{name -> CHARACTER
 member_of -> function
 comment ->
     "     (STRING1 NUMBER1) => STRING
     "
     "     Return a single-character STRING containing
     "the NUMBER1'th character of STRING1.  If NUMBER1
     "< 0, count from the end of STRING1."
 n_arguments -> 2
 algorithm -> ^C_CHARACTER_FUNCTION
 }
```

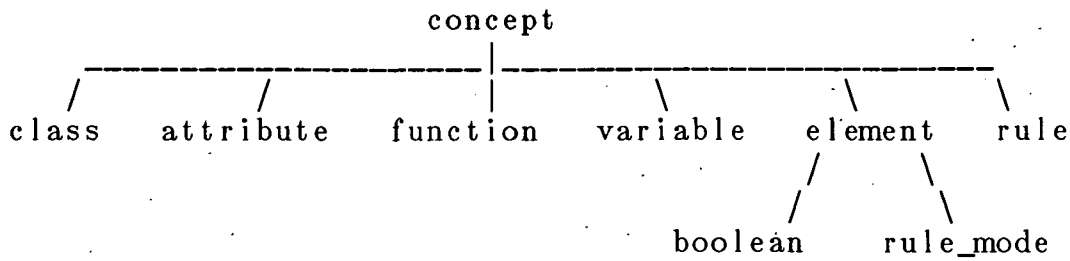
```
    test(TANGENT)          $ Use of "test" to determine
                           $   the non-existence of a
{name -> NIL               $   particular named RECORD.
 member_of -> element
 }
```

Following a certain measure of experience with STAR, the programmer will discover that STAR generates new named RECORDs not only when a previously unused reference name is entered, but as well when a reference name is simply misspelled, as, for instance, if "adddfz" were entered in place of "add" in the above example. It is usually the programmer's first instinct to try to purge the system of such unneeded named RECORDs immediately following their accidental creation; however, it is recommended simply to ignore these ineffectual entries. As will be seen later, the "save" command, which stores the contents of the STAR knowledge base in text format on a file, simply passes over named RECORDs which are not referenced elsewhere within STAR, and thus the life of such accidentally created named RECORDs is limited to a single STAR programming session.

## 8. Structure of the Network

The knowledge base of STAR is a highly structured organization of items designed around the concept of a semantic network. The term "semantic network" is used with some diversity in the literature, and thus it is useful to give a more precise description here. The "nodes" in the STAR knowledge base are individual named RECORDs (each with a unique name), representing quantities involved in a particular application. The "links" are the attribute-value pairs within named RECORDs. Each link is thus labeled by the attribute used, and directed from the named RECORD to the UNIT appearing as the value of that attribute. The UNIT as the value of a link need not be a named RECORD but might be of any type, such as a LIST of named RECORDs. This is consistent with a number of other Artificial Intelligence languages, but is not always considered basic to the "semantic network" formalism.

In addition, the STAR knowledge base is organized into a hierarchy by two relations, (1) subclasses within classes of quantities, and (2) individual members within classes. For example, the named RECORD "true" is a *member* of the class "boolean", and "boolean" is a *subclass* of the class "element", and so forth. At the top of the hierarchy resides the named RECORD "concept", which is a class covering six subclasses, "class", "attribute", "function", "variable", "element" and "rule", each of which covers further subclasses and members. The following diagram depicts the initialized hierarchy of classes and subclasses in STAR. This hierarchy may be augmented but not altered or reduced by definitions relating to a specific application.




Classes and attributes are used in construction of the knowledge base itself, functions and variables facilitate conventional symbolic programming, and rule-based operation is possible through the definition and invocation of sets of rules, with members of the class "element" representing remaining quantities in the domain of the application.

Largely omitted in the above diagram are the named RECORDs which appear as individual members of the illustrated classes. Each class is itself cross-listed as a member of the class "class", this in addition to its subclass relation as illustrated above. Members of the class "attribute" include the named RECORDs "name", "member_of", "members" (links a class with a LIST of members), "subclass_of", "subclasses" (links a class with a LIST of subclasses) and so forth, all used extensively in the attribute-value pairs of RECORDs throughout the STAR knowledge base. The class "function" contains all of the functions inherent to the language STAR such as "add", "character", "select", and so forth. As well, the classes "variable", "element", "boolean" and "rule_mode" contain other named RECORDs which will be described in the context of their use. As an application system is developed, new members of all classes may be defined to aid in the operation of the final system.

The above description of the organization and layout of the STAR knowledge base is necessarily schematic in nature. A simple way to become acquainted with the details of the initialized STAR knowledge base and its organization is to enter reference names of key RECORDs in this hierarchy to see how these concepts are related to other concepts.

Two built-in functions which may aid in viewing the semantic network (these functions have other uses as well) are "enumerate" and "path". The function "enumerate" takes a class as an argument and returns a LIST containing the members explicitly listed for that class as well as the members of all subclasses of that class. Thus the EXPRESSION "enumerate(element)" evaluates to a LIST of all direct members of the class "element" plus all members of "boolean", all members of "rule_mode" and so forth. The function "path" returns a LIST starting with "concept" and proceeding through subclasses up to the named RECORD provided as its argument. In this manner, "path" describes the position of a particular named RECORD within the hierarchy of subclasses and members.

## 9. Basic Use of the Network

Section 7 listed one major distinction between named RECORDs and unnamed RECORDs -- the use of reference names with named RECORDs. Another major distinction concerns the types of built-in operations available for each. RECORDs of all types may be operated upon by the functions "get", "put", "omit", and so forth. These functions are "nondestructive", however, in that they never alter the RECORD supplied as an argument; rather, any modifications are performed on a copy of the RECORD.

In contrast to this, STAR does allow the direct modification of user-defined named RECORDs, and to some extent, the built-in named RECORDs of STAR as well. This is the manner in which a programmer or program effects changes in the STAR knowledge base. The primary restriction is that no modifications may be made to initialized named RECORDs (such as "class", "attribute", "add", "member_of", etc.) such that particular entries are to be changed from their initialized state. An attempt to perform such modifications results in a PERMISSION ERROR. Otherwise, additions may be made to the initialized named RECORDs, the additions themselves may be modified at will, and user-defined named RECORDs may be modified in whatever manner suits the application, given the limits of the STAR built-in functions for operations on named RECORDs.

The three central functions for performing direct modifications on named RECORDs are "assert", "retract" and "modify". Their use is illustrated in the example below.

**Example 11:**

```
   $ Assume previously defined: "block", "height" and
   $   "block_1".

    block_1                      $ Accessing the named
                                 $   RECORD "block_1".
{name -> BLOCK_1
 member_of -> block
 }

    assert(block_1 height 7)     $ Adding a new attribute-
                                 $   value pair to the def-
{name -> BLOCK_1                 $   inition of "block_1".
 member_of -> block
 height -> 7
 }

    block_1                      $ The resulting status of
                                 $   "block_1".
{name -> BLOCK_1
 member_of -> block
 height -> 7
 }
```

```
   modify(block_1 height 8)      $ Changing an existing
                                 $   attribute-value pair.
{name -> BLOCK_1
 member_of -> block
 height -> 8
 }

   retract(block_1 height)       $ Removing an attribute-
                                 $   value pair.
{name -> BLOCK_1
 member_of -> block
 }
```

The functions "assert", "modify" and "retract" overlap in some respects. The "assert" function may only be used to add a value for a previously excluded attribute, while "modify" may be used for this purpose or to alter the value of an included attribute. The operation of "modify" is similar to a "retract" operation followed by an "assert" operation (this is more important in the context of the next section).

It is conceivable that one could enter the definition of a named RECORD first by creating it through entering a reference name or using the "locate" function, followed by using the "assert" or "modify" function several times in succession, once for each attribute to be included. Two shortcuts to this process are the built-in functions "define" and "create".

The function "define" takes a completely formed named RECORD, and enters it as a whole into the STAR knowledge base. The named RECORD may exist prior to this, but no conflicts are allowed between values already existing for particular attributes and values supplied in the new RECORD. This is as if the function "assert" were used in entering each attribute-value pair of the RECORD. The function "create" allows a named RECORD with membership information only to be created directly in a simple manner. The use of these two functions is illustrated below.

**Example 12:**

```
$ Assume previously defined: "city", "state",
$   "climate" and "hot_spots".

define(.                           $ Entering an entire
 {name -> LOS_ANGELES              $   named RECORD using
  member_of -> city                $   "define".
  state -> california
  climate -> mild
  hot_spots ->
      [venice_beach disneyland westwood]
  })
```

```
{name -> LOS_ANGELES
 member_of -> city
 state -> california
 climate -> mild
 hot_spots -> [venice_beach disneyland westwood]
 }

    create(pasadena city)              $ Using "create" to enter
                                       $   "name" and "member_of"
{name -> PASADENA                      $   values only.
 member_of -> city
 }
```

The RECORD provided as an argument to "define" must contain an entry for "name" but is not required to contain additional entries. At some point, however, it is strongly recommended that each named RECORD be provided with an entry for "member_of", as without this entry a named RECORD is not accessible through the subclass/membership hierarchy of the STAR knowledge base and is overlooked by the function "save" as it stores the STAR knowledge base in text form in a file.

Two additional functions, "revise" and "merge", operate in an analogous manner to the non-destructive functions "detach" and "attach" described in Section 4. The function "revise" takes two arguments, a named RECORD and a LIST of attributes, and revises the named RECORD to contain only entries for the attributes of the LIST, in the order given. The function "merge" works the other way, taking two RECORDs, the first of which must be a named RECORD, and augmenting that RECORD to include values for all attributes included in the second RECORD but not the first.

The above described functions, "assert", "modify", "retract", "define", "create", "revise" and "merge", are the primary built-in functions for performing general operations on named RECORDs in the semantic network. A few additional functions are described in the following sections and involve specialized operations such as those relating to the use of variables. In all cases, the modifications are performed directly on the named RECORDs supplied as arguments to the functions, and any attempts to modify directly the initialized portions of named RECORDs in the initialized STAR knowledge base result in PERMISSION ERRORs.

Before proceeding to the more advanced semantic network mechanisms of STAR, a note about general design principles is in order. In Section 8, mention is made of "forward referencing" as present in STAR, where the reference name for a particular named RECORD may be used before the entire definition of the named RECORD is in place. STAR simply sets up a single-entry named RECORD, containing only a "name" specification, and waits for the remainder of the definition to be entered.

This device does not generalize, however, to mean that a named RECORD may be *used* in the capacity for which it is intended before the essentials of its definition have been entered. An example is that of specifying the members of a class. While it is permissible to use the name of a parent class before it is defined, STAR does not allow individual members of the class to be specified before the parent is, in fact, defined as a class.

Analogously, an attribute must be defined as an attribute before it is used in a RECORD (some exceptions to this rule are made for simplicity, but it is nevertheless a good practice), a function or rule must be defined before it is applied, and a variable must be defined as a variable before being given a set of bindings. In the case of classes and their members or subclasses, the restriction results from a set of automatic side-effects generated in STAR whenever values for "member_of" or "subclass_of" are specified. The automatic side-effects insert reciprocal entries in the "members" and "subclasses" entries of parent classes and encompassing classes. For this to occur, however, the parent or encompassing classes must exist prior to the definition of individual members or subclasses.

The effect of the above-mentioned "definition-before-use" restriction is that the entry of definitions pertaining to classes and members within an application system must proceed in a "top-down" or "general to specific" order. Note, however, that this does not mean the application system itself must be developed in a top-down manner: simply that once it is developed, the loading process by which the definitions are entered into STAR must proceed as such, from general classes to specific classes to individual members. Various error messages result if the definitions are loaded or entered in the wrong order. (The "save" function, which stores the definitions in the STAR knowledge base in text form in a file, does this in a top-down order so that a subsequent loading operation is possible. Section 12 describes "save" and associated functions.)

## 10. Aspects and Side-effects

The knowledge base of STAR can be used in a number of different ways, from a simple repository of named RECORDs to a highly interdependent association of implicit and explicit descriptions of quantities. Some of the more complex ways of using the knowledge base are covered in this section. For the reader not interested in these particular features of STAR, the section may be safely omitted without loss of continuity.

Two basic notions are involved in the discussion below: (1) the inheritance of attribute-value pairs by class-members from their respective parent classes, and (2) the automatic generation of side-effects stemming from asserting or retracting values in named RECORDs. The mechanisms involved with inheritance are described first.

### Inheritance

Basically, the information stored in a named RECORD representing a class is of a different sort from the information stored in individual members for that class. Associated with the class may be relations concerning the class as a separate entity and not pertaining to any individual member of the class. On the other hand, associated with each member of the class is information which applies distinctly to that member.

In some cases, however, it is useful for general information which applies to all members of a class to be stored centrally in the class description itself and to be available implicitly rather than explicitly to the members of the class. In such cases, the "member" information stored at the class level should be segregated from the "class" information

stored at that level. STAR accomplishes this through the use of the attribute "pattern". The attribute "pattern" associates a class with an unnamed RECORD containing attributes and values which may be inherited by individual members of that class.

It is useful to note that in STAR, the inheritance mechanism does not work automatically, but must be explicitly invoked. For example, use of the "get" function on a member of a class may retrieve only the attribute-value pairs explicitly stored within that RECORD. Use of a different function, "determine", invokes the inheritance mechanism, giving priority to explicitly stored attribute-value pairs in a RECORD, but otherwise relying on "patterns" stored within parent classes to supply remaining attribute-value pairs. The following example illustrates the use of the "pattern" attribute in conjunction with the "determine" function. Information concerning the objects described is stored both explicitly (within the definitions of particular individuals) and implicitly (within the definition of the parent class).

**Example 13:**

```
$ Assume previously defined: "automobile",
$   "flammable_liquid", "wheels", "fuel", "medium",
$   "make", "model", "year", "color", "abundance",
$   "use", "auto_1" and "gasoline".

   automobile                  $ Display the definition for
                               $   the class "automobile".
{name -> AUTOMOBILE            $   Implicit information for
 member_of -> class            $   members of "automobile"
 subclass_of -> element        $   stored under "pattern".
 members -> [auto_1 auto_2]
 subclasses -> []
 pattern ->
     {wheels -> 4
      fuel -> gasoline
      medium -> [street highway]
      }
}

   auto_1                      $ Display the definition for
                               $   the individual "auto_1".
{name -> AUTO_1
 member_of -> automobile
 make -> volkswagen
 model -> beetle
 year -> 1974
 color -> blue
 }
```

```
   get(auto_1 year)          $ Retrieving a value stored
                             $   explicitly within the
1974                         $   named RECORD "auto_1".

   get(auto_1 fuel)          $ Attempt to use "get" for
                             $   an implicitly-stored
{name -> NIL                 $   value.
 member_of -> element
 }

   determine(auto_1 year)    $ Use of "determine" to
                             $   retrieve an explicitly-
1974                         $   stored value.

   determine(auto_1 fuel)    $ Use of "determine" to
                             $   retrieve an implicitly-
{name -> GASOLINE            $   stored value.
 member_of -> flammable_liquid
 abundance -> very_common
 use -> [automobile motorcycle boat]
 }
```

The above example illustrates the simplest type of inheritance within STAR. Implicit values for attributes may be stored under the "pattern" attribute in any class, and these values may be be accessed for all members of such classes or members of subclasses of the classes.

Another mechanism which works in conjunction with the "pattern" attribute is the notion of "aspects". This is a variation on the semantic network principle in which an attribute is not always associated with simply a "value", but may have several "aspects" to consider, of which the value is only one aspect. Other aspects may be computational procedures describing what to do in order to calculate the status of the value, computational procedures describing side-effects to be taken when a new value is added for the attribute or an old value is retracted, and so forth.

In STAR, there are five built-in aspects: "value", "default", "if_needed", "if_asserted" and "if_retracted". Other aspects may be defined in the course of developing an application system. The aspects "default" and "if_needed" are used in the inheritance of values from classes to individuals, "if_asserted" and "if_retracted" are used in generating side-effects to modifications made in the knowledge base, and "value" is used to distinguish the actual value from other aspects when more that one aspect is present.

The five aspects are implemented as attributes in the hierarchy of named RECORDs, as their use is as attributes in a special type of RECORD. This RECORD, which may be substituted for the value of an attribute in a named RECORD, has the form

```
{aspects ->
    {aspect_1 -> value_1
     aspect_2 -> value_2
     ...
     }
},
```

where "aspect_1" and "aspect_2", etc., are individual aspects and "value_1" and "value_2", etc., are their values.

The following example illustrates the construction used in STAR to represent different aspects of an attribute within a RECORD. The use of three of these aspects, "value", "default" and "if_needed", is presented in conjunction with the three STAR functions "determine", "estimate" and "calculate".

**Example 14:**

```
    $ Assume previously defined: "bolt", "object",
    $  "use_type", "use", "threads_per_inch",
    $  "count_threads", "length_shaft", "bolt_1" and
    $  "fastening".

    bolt                          $ Definition and "pattern"
                                  $  of "bolt". The "use"
{name -> BOLT                     $  attribute has only the
 member_of -> class               $  aspect "value" (this
 subclass_of -> object            $  could be simplified to
 members -> [bolt_1]              $  "use -> fastening"),
 subclasses -> []                 $  and "threads_per_inch"
 pattern ->                       $  has aspects "default"
    {use ->                       $  and "if_needed".
        {aspects ->
            {value -> fastening
             }
        }
     threads_per_inch ->
        {aspects ->
            {default -> 20
             if_needed ->
                /(
                 count_threads()
                 length_shaft()
                 )
             }
        }
    }
}
```

```
    bolt_1                      $ Definition for "bolt_1"
                                $   (no explicit values for
{name -> BOLT_1                 $   "threads_per_inch" or
 member_of -> bolt              $   "use").
 }

    determine(                  $ Since no "value" aspect
     bolt_1                     $   for "use" is explicitly
     use                        $   stored in "bolt_1", the
     )                          $   value from "bolt" is
                                $   returned.
{name -> FASTENING
 member_of -> use_type
 }

    estimate(                   $ "estimate" looks for
     bolt_1                     $   an explicit or
     threads_per_inch           $   implicit "default"
     )                          $   aspect.

20

    calculate(                  $ "calculate" looks for
     bolt_1                     $   an explicit or
     threads_per_inch           $   implicit "if_needed"
     )                          $   aspect, also evaluating
                                $   the result.
16
```

Note that while the aspects "value", "default" and "if_needed" may all be present in describing a particular attribute, their usage does not overlap with each other. The "determine" function searches only for an explicit or implicit "value" aspect. Likewise, "estimate" applies only to "default" and "calculate" only to "if_needed", with "calculate" additionally evaluating its result. This separation allows the programmer to set up individualized searching strategies through calling "determine", "estimate" and "calculate" in various combinations and orderings. For example, one such strategy might call "determine" first, and if that fails, "calculate", thus giving priority to the "value" aspect over the "if_needed" aspect and bypassing the "default" aspect altogether.

There are a few subtle distinctions intended in the use of the "value", "default" and "if_needed" aspects. The "value" aspect is intended to be used in specifying only some property of an object or members of a class which may always be said to hold. (Thus, an implicit "value" aspect would not normally be overridden by a distinctly different explicit "value" aspect). In contrast, the "default" aspect is intended to be merely a suggestion, to be used only in estimating the properties of an object. An implicit "default" aspect may well be overridden at the explicit level or at a closer implicit level, where more specific knowledge is stored concerning the object in question. The "if_needed" aspect is

somewhat distinct from both "value" and "default" in that the obtained data value is evaluated before it is returned. This allows the storage of EXPRESSIONs for calculating various properties.

As an example in the distinction between "value", "default" and "if_needed", consider the problem of determining various properties belonging to a human individual described in the knowledge base.. Since all humans should unquestionably possess a brain, this property may be stored using the "value" aspect at the level of the class "human" (or "animal", if present above "human"). Other items of less critical importance such as eyes, teeth, arms and legs might better be stored using the "default" value, since it cannot unquestionably be assumed that these are present. Finally, highly variable yet determinable properties such as age, height, weight and so forth might be stored either as explicit values, if these are known, or implicit procedural methods for calculating the data values, with these being stored using the "if_needed" aspect at the class level.

One final word on the inheritance of values between named RECORDs. It is possible to examine a particular inherited aspect directly by using the function "obtain". This function takes three arguments, a named RECORD, an attribute and a second attribute which is an aspect ("value", "default", "if_needed", "if_asserted", "if_retracted" or a user-defined aspect in use for a particular application system). The function "obtain" searches up the path starting at the named RECORD in question and proceeding to the class "concept" and returns the first data value associated with the specified attribute and aspect. For instance, using the above example as a basis, the EXPRESSION

obtain(bolt_1 use value)

would return the named RECORD "fastening", while the EXPRESSION

obtain(bolt_1 threads_per_inch if_needed)

would return the EXPRESSION "/(count_threads() length_shaft())".

## Side-effects

The aspects "if_asserted" and "if_retracted" are associated with a separate mechanism for generating side-effects to modifications made in the STAR knowledge base. These side-effects most often concern the maintenance of secondary or reciprocal links within the knowledge base. For example, each named RECORD contains an entry for the attribute "member_of", linking that named RECORD with a second named RECORD which is its parent class. In turn, the named RECORD for the class contains an entry for the attribute "members", which specifies a LIST of all members of that class. It is useful to have all modifications involving "member_of" automatically invoke corresponding modifications for the "members" attribute at the class level, and so forth.

Such side-effects are built-in with regard to the attributes "name", "member_of"..."members", and "subclass_of"..."subclasses" in STAR. Modifications involving the "name" attribute are automatically reflected in the maintenance of a

directory which is used to locate individual named RECORDs given their reference names. As well, modifications dealing with the attributes "member_of" and "subclass_of" automatically generate modifications in entries for "members" and "subclasses" within the target classes. The mechanism by which these processes occur is implicit for the attributes "name", "member_of" and "subclass_of", but explicit for all other attributes.

Several components are necessary in setting up an automatic side-effect. To start with, the entry "side_effects -> true" must be placed within the defining RECORD of any attribute for which a side-effect is desired. This adds considerable efficiency to the operation of STAR as it is thus necessary to check for possible side-effects only in a few cases and not every time a value is asserted or retracted in the knowledge base.

The second component involves the aspects "if_asserted" and "if_retracted". The aspect "if_asserted" is used to provide a sequence of steps to be taken whenever a new value is provided for a particular attribute within a particular named RECORD. This is not restricted solely to the use of the function "assert", but may be the case when using any of the functions "assert", "modify", "define", "create" or "merge", as well as other specialized functions which perform modifications to the knowledge base. Likewise, the aspect "if_retracted" is used to provide a sequence of steps to be taken whenever a value is retracted from a named RECORD in the knowledge base. This may occur through the use of "retract", "modify", "revise", and so forth. Note that the function "modify" when used to change an existing value for a particular attribute actually performs the equivalent of a "retract" operation followed by an "assert" operation.

Entries for "if_asserted" and "if_retracted" are made in the "pattern" descriptions for various classes under which side-effects are to occur. Usually, just one set of entries is made under the top class "concept", and the side-effects prescribed by these entries apply to all named RECORDs with regard to the attribute in question. If particular side-effects are to apply only to a subset of all named RECORDs, then the appropriate entry is made at the level of a subclass of "concept" or even an individual member of a class. As an example, the following "pattern" entry might be placed in the named RECORD "concept" for performing the side-effects related to a particular user-defined attribute "specialization_of".

```
pattern ->
    {specialization_of ->
        {aspects ->
            {if_asserted -> [to_specializations]
             if_retracted -> [from_specializations]
             }
        }
    }
```

Two aspects are provided for the attribute "specialization_of" in this illustration. The "if_asserted" aspect provides a LIST of functions to call whenever a value is added for "specialization_of" within any named RECORD, and the "if_retracted" aspect provides a similar LIST of functions to call whenever a value is retracted for "specialization_of" within any named RECORD.

Each function in the LIST associated with "if_asserted" or "if_retracted" ("to_specializations", etc., above) takes one argument which is the named RECORD being modified, this taken after the modification when asserting and before the modification when retracting. The functions called may then look into the named RECORD provided as an argument to retrieve the value being added or deleted, acting accordingly.

For the attributes "name", "member_of" and "subclass_of", the side-effects generated concerning the internal directory of named RECORDs and "members" and "subclasses" entries have been discussed in the above. For these attributes as well, a fairly rigorous testing is carried out for all values to be asserted or retracted before the actual operations are carried out. This prevents cases where a new value has been asserted, for example, only to result in an error detected while executing the side-effects of the modification. For side-effects defined for other attributes, similar checks should be made before asserting new values, as errors detected while carrying out generated side-effects occur *after* the actual modification and thus the value remains in its asserted state even though the STAR error chaining has been invoked and the side-effect operation abandoned. For the function "define", for example, it may be the case that several complex side-effects have been generated in the asserting of values within a given RECORD, only for the last entry to fail due to an error detected during its side-effect operation. At this point, it is too late to undo the side effects generated by previous entries in the RECORD. To avoid such inconsistencies in the STAR knowledge base, values to be asserted for attributes with side-effects should always be checked closely before proceeding with the actual operation.

## Part III: Programming in STAR

### 11. Variables and Functions

With the organization of STAR covered in Part II, it is now possible to proceed with a description of actual programming in STAR. In a general sense, "programming" in symbolic languages means any mechanism by which procedural information is stored within the language so that it may be retrieved and executed at a later point in time. Thus, the placement of an EXPRESSION as the value of a particular attribute in a named RECORD qualifies as "programming", since that EXPRESSION may be retrieved and evaluated at a later time, thereby executing the stored command to the STAR interpreter.

In addition to the many subtle variations on the general sense of programming available in STAR, two large-scale mechanisms are provided. The first concerns the definition of variables and functions, similar to the type of programming found in general-purpose languages like C and PASCAL. This first type of programming is covered in Sections 11 through 14. The second mechanism involves variables and rules. Using this mechanism, the programmer can describe the operation of an application in the form of a production system. Section 15 describes the use of this second mechanism.

In STAR, variables and functions are implemented as named RECORDs belonging to the classes "variable" and "function", respectively. Certain restrictions are placed on the exact form of the RECORDs contained within these classes. Variables are covered first.

#### Variables

A variable is a named RECORD which is a member (directly or indirectly) of the class "variable" and which contains an entry mapping the attribute "bindings" to a LIST of values (the first element in this LIST is the current value). The named RECORD below is an example of the definition for a hypothetical variable "i".

```
{name -> I
 member_of -> variable
 bindings -> [44 3]
 }
```

It is important to note that when the reference name for a variable is entered (e.g., "i" for the variable above), one gets the entire definition of the variable and not simply the value, as occurs in a number of other programming languages. The value of a variable is accessed by the "dot" function, shown in standard and abbreviated form for the variable "i" below.

```
dot(i)  <==>   .i
```

The use of explicit measures for accessing the value of a variable instead of implicit measures (automatic substitution of a variable name with its value) has several advantages. First of all, by this method it is always clear whether a variable is indicated or its value is indicated. In a LIST of arguments for a function or in the context of passing a variable name as an argument, only the reference name of the variable is used; in contexts where the value is desired, the "dot" function is used along with the reference name. Secondly, it is not necessary to "protect" variable names with the "quote" function in contexts where they might be evaluated (e.g., when assigning a new value to a variable). Evaluation of a variable itself produces no effect, similar to evaluation of any named RECORD.

As mentioned above, each variable is associated not with a single value, but rather with a LIST of values. This LIST behaves like a stack and is used for "dynamic scoping" of values. When a function is called which designates a particular variable as having local significance, a new binding is created for the variable, protecting any previously designated value so that it is once again available after the function completes. This allows a recursive function to reuse the same variable, as a new binding is inserted into the variable's bindings LIST for each call to the function. As a function completes, each variable designated as having local significance is stripped of its current binding by removing the first element in its bindings LIST.

The process of adding and removing bindings for local variables to functions occurs automatically. The mechanism which does this is described more fully in the context of function definitions, later in this section. In addition to this, it is also possible to add and remove bindings of a variable manually through the use of the built-in functions "new" and "old". The use of these functions is described below, along with the built-in function "set", used to alter the current value of a variable.

**Example 15:**

```
define(                          $ Define a variable "i"
 {name -> I                      $   with no bindings.
  member_of -> variable
  bindings -> []
  }
 )

{name -> I
 member_of -> variable
 bindings -> []
 }

 new(i 100)                      $ Create a new binding
                                 $   with a value of "100".
```

```
{name -> I
 member_of -> variable
 bindings -> [100]
 }

    .i                          $ Current value of "i".

100

    new(i 200)                  $ Create a new binding
                                $   with a value of "200".
{name -> I
 member_of -> variable
 bindings -> [200 100]
 }

    .i                          $ Current value of "i".

200

    set(i 500)                  $ Change the current value
                                $   to "500".
{name -> I
 member_of -> variable
 bindings -> [500 100]
 }

    .i                          $ Current value of "i".

500

    old(i)                      $ Remove the current value
                                $   of "i", returning to
{name -> I                      $   its next binding.
 member_of -> variable
 bindings -> [100]
 }

    .i                          $ Current value of "i".

100
```

If the function "new" is called for a variable with no "bindings" entry, an appropriately formed entry is first set up (i.e., "bindings -> []") before the given value is inserted as a new binding. User-defined functions treat variables with missing "bindings" entries in a similar manner and thus it is sufficient to define variables simply using the "create"

function. (Note, however, that "set" may not be used until a variable is specified as having at least one binding: this either by original definition, through use of "new", or through the calling of a function which creates a binding for the variable.)

In practice, the functions "new" and "old" are not used nearly as often as the function "set", which is the standard method used for updating the current value for a variable. Usually, the adding and removing of bindings is performed primarily in the automatic sense in conjunction with function applications. The functions "new" and "old" are still available, however, for uses which fall outside this context.

Since variables are themselves named RECORDs and can be used as values in various contexts, it is sometimes the case that the value of one variable is in fact another variable. This sort of indirection occurs when a variable is passed as an argument to a function. The following example illustrates the use of variables in this sense.

**Example 16:**

```
    define(                     $ Define a variable "i",
     {name -> I                 $   initializing it with
      member_of -> variable     $   a single binding to
      bindings -> [100]         $   the value "100".
      }
     )

{name -> I
 member_of -> variable
 bindings -> [100]
 }

    define(                     $ Define a variable "j",
     {name -> J                 $   initializing it with
      member_of -> variable     $   a single binding to
      bindings -> [i]           $   the variable "i".
      }
     )

{name -> J
 member_of -> variable
 bindings -> [i]
 }

    .i                          $ Current value of "i".

100

    .j                          $ Current value of "j".
```

```
{name -> I
 member_of -> variable
 bindings -> [100]
 }

   ..j                          $ Value of the value of
                                $  "j", equal to current
100                             $  value of "i".
```

In applications such as the above in which one variable takes on another variable as its value, it is important to distinguish between those times when simply the name of the referenced variable is desired (one application of "dot") and those times when the value of the referenced variable is desired (two applications of "dot").

The initialized knowledge base for STAR contains three variables, "pound_sign", "control" and "alternatives", located in the class "variable". The latter two are described in Section 15 in the context of rule-based operation. The variable "pound_sign" implements the mechanism by which the pound-sign ("#") escape sequence is operated. At the completion of each execution cycle of user entry, evaluation and subsequent display of the resulting UNIT, the resulting UNIT itself is stored as the new current value of the variable "pound_sign". This value may be retrieved during the next execution cycle by evaluating the EXPRESSION ".pound_sign" or simply by using the abbreviated escape sequence "#".

A few additional points concerning variables must be discussed before moving on to functions. In STAR, variables have an existence of their own as named RECORDs, independent of the definition of functions and other quantities in an application. This has several ramifications. First of all, since each named RECORD has a distinct name, the names used for variables may not be used for other quantities elsewhere in an application. Thus, each variable is global in terms of its definition, while it may possess local bindings within the application of individual functions. Typically, the programmer will reserve a set of names (say the single-letter names "a" through "z") as general-purpose variables and create other special-purpose variables as needed, giving these names which indicate their use as variables (e.g., "number_of_items"). This practice minimizes the occurrence of using a particular name twice, once as a variable and once as some other quantity (the STAR interpreter will print an error message upon receiving the second definition).

Secondly, the use of a variable as an argument name or local variable within a function is independent of the definition of the variable itself. Thus, when creating an application in STAR, the programmer must explicitly enter definitions for all variables, even those used only locally within particular functions.

## Functions

There are two related forms for defining functions in STAR, one for functions whose algorithms are really defined in other languages such as FORTRAN, C and PASCAL, and one for functions whose algorithms are defined in terms of STAR commands. These two forms may be used interchangeably in defining individual named RECORDs contained directly or indirectly within the class "function". The first form (external functions) is covered only briefly in this section, with details appearing in Part IV of the Tutorial Guide.

The built-in commands of STAR are implemented as routines written in the language C, and thus their definitions reflect the first form. The following is a listing of the definition for the built-in function "add".

```
{name -> ADD
member_of -> function
comment ->
    "    (NUMBER1 NUMBER2) => NUMBER
    "
    "    Addition."
abbreviation -> "+"
n_arguments -> 2
algorithm -> ^C_ADD_FUNCTION
}
```

The built-in commands of STAR all appear as direct members of the class "function". The attributes listed under each built-in function include "comment", "abbreviation" (optional), "n_arguments" and "algorithm". The attribute "comment" provides a STRING describing the syntax and operation of the function. The attribute "abbreviation", if present, provides a single-character STRING containing a symbol which may be used for the function name in an abbreviated form. The attribute "n_arguments" specifies the number of arguments which must be sent to the function. Finally, the attribute "algorithm" provides a CONNECTION pointing to the routine which actually does the operation specified for the function.

It is also possible to define other functions external to STAR using this format. This allows the programmer to (1) extend the built-in command set of the STAR interpreter by writing new functions in C which manipulate UNIT structures, and (2) include application-specific routines defined in other languages which operate on data structures also defined in those languages and passed as arguments in the form of STAR CONNECTIONs (see Part IV of the Tutorial Guide).

The other form for function definitions, the one used for symbolic functions within STAR, employs a different set of attributes. The following is an example of the definition and use of a simple function for calculating the average of two numbers.

**Example 17:**

```
    define(
     {name -> AVERAGE
      member_of -> function
      comment ->
          "     (NUMBER1 NUMBER2) => NUMBER
          "
          "     Returns the average of two NUMBERs."
      arguments -> [a b]
      temporary -> []
      algorithm -> [return(/(+(.a .b) 2))]
      }
     )

{name -> AVERAGE
 member_of -> function
 comment ->
     "     (NUMBER1 NUMBER2) => NUMBER
     "
     "     Returns the average of two NUMBERs."
 arguments -> [a b]
 temporary -> []
 algorithm -> [return(/(+(.a .b) 2))]
 }

    create(a variable)                $ Define argument
                                      $   variables.
{name -> A
 member_of -> variable
 }

    create(b variable)

{name -> B
 member_of -> variable
 }

    average(3.3 3.7)

3.5

    average(-12 24)

6
```

```
return(100)                    $ (See explanation
                               $  for this below.)

{return_value->100}
```

There are a number of differences between the two forms for defining functions. The primary difference is that the "algorithm" attribute is not associated with a CONNECTION to an external routine, but rather with a LIST of UNITs specifying the steps in the execution of a function. In the above example, there is only one step, specifying a particular value to return.

Another difference is that the "n_arguments" (number of arguments) attribute for external functions is replaced by the attribute "arguments". In the context of external functions, it is only necessary to know how many arguments to send; here, it is also necessary to know which variables are to be used as hosts to the arguments. Thus, the attribute "arguments" provides a LIST of variables to be bound one-to-one with the incoming arguments. Related to this, the attribute "temporary" provides a second LIST of variables having local significance to the function. Variables appearing in either the "arguments" LIST or "temporary" LIST are given an additional binding for the duration of the function execution (to be removed following completion of the function execution). If no "bindings" entry exists within a variable's definition, one is created prior to the insertion of the new value. Each "temporary" variable is given a new binding initialized to the value "nil", while each variable listed under "arguments" is given a binding to the corresponding argument sent to the function.

In user-defined functions (functions of this second form), entries for the attributes "arguments" and "temporary" are both considered optional. In the case of a missing "arguments" entry, the STAR interpreter assumes no arguments. Likewise, a missing "temporary" entry signifies the absence of local bindings for variables.

An additional difference between functions defined in the second form as opposed to the first form is that it is not permitted to apply these functions using the abbreviated forms for EXPRESSIONs. Thus no entry for the attribute "abbreviation" appears in the definitions for functions of this variety.

In the above example, the built-in function "return" was introduced. This is a function used exclusively for specifying the values to be returned by user-defined functions in STAR. The traditional implementation of such functions as "return" (or "break", for exiting an iterative loop) gives them a special status of having the power to affect control decisions outside the scope of their evaluation. For example, "return" actually would make a control decision at the level of evaluating the LIST of UNITs for the function in which it appears, and so forth.

The decision was made in STAR that this should not be so; rather, functions like "return" (also "result", "break", "skip" and "stop", introduced in later sections) behave the same as other functions, evaluating their arguments and returning particular UNITs as results to the next outer level of evaluation. The distinction is that these functions return UNITs which are of a special form to be recognized by outer levels as triggers for control decisions which may be made only at those outer levels. Thus, the function "return", as indicated in the example above, returns a special RECORD containing the built-in attribute "return_value" and a value equal to the UNIT to be returned. The

STAR interpreter evaluates sequentially the UNITs in the algorithm LIST of the function, and whenever a RECORD of this form appears as the result of evaluating a particular UNIT, further evaluation of the algorithm LIST is halted and the specified value returned. There will be further examples of this and other related functions in the remaining sections.

This completes the initial description of functions in STAR. As the remaining constructs related to programming are introduced (input/output, logical operations and general programming structures in the following sections), further examples of user-defined functions are provided.

One related operation which may be described at this point is the use of the built-in function "apply". This function is quite similar to the corresponding function "apply" in LISP. The function takes two arguments, here a function as the first argument and a LIST of unevaluated arguments as the second argument. The function "apply" then forms an EXPRESSION involving the designated function and arguments and evaluates that EXPRESSION, returning the result of evaluation.

The operation of "apply" is illustrated in the example below, building on the definition of the function "average" as in Example 17. The function "apply" has several uses, allowing the programmer to specify the application of a function determined at run time to a given set of arguments, and also allowing functions defined externally to STAR to call user-defined functions within STAR (this is covered more fully in Section 16).

**Example 18:**

```
    apply(average [3 4])        $ Use of "apply" with a
                                $   user-defined function.
3.5

    apply(add [*(10 4) 5])      $ Arguments may be EXPRES-
                                $   SIONs to be evaluated.
45
```

## 12. Input/Output and Files

The input and output functions in STAR are roughly divided into two general categories: functions which communicate with the user terminal, and functions which communicate with files. Each of these categories is further subdivided into functions which deal primarily with UNITs and functions which deal primarily with text contained in STRINGs.

The functions relating to communication between STAR and the user terminal are "parse", "display", "input" and "output". An additional function, "format", provides a capability for intermixing text and UNITs within a display line in a simple manner. The operation of these five functions is illustrated in the example below, first by themselves and then in the context of a user-defined function.

**Example 19:**

```
    parse()                          $ Accept a UNIT as input.
+(2 3)                               $ <--- UNIT entered by user.

+(2 3)                               $ Return value (unevaluated!).

    display(member_of)               $ Display the UNIT "member_of".
{name -> MEMBER_OF                   $ <--- Displayed UNIT.
 member_of -> attribute
 comment ->
    "    The immediate parent class of a named
    "RECORD.  Asserting or retracting values for
    "'member_of' has automatic side-effects in the
    "'members' entry for the parent class."
}

{name -> MEMBER_OF                   $ Return value of "display".
 member_of -> attribute
 comment ->
    "    The immediate parent class of a named
    "RECORD.  Asserting or retracting values for
    "'member_of' has automatic side-effects in the
    "'members' entry for the parent class."
}

    input()                          $ Input text from the user.
Move two units to the left.          $ <--- Text entered.

"Move two units to the left.         $ Resultant STRING (note
""                                   $   final carriage-return).

    output(                          $ Print text contained
    "Find a path to point A.         $   within a STRING.
    "")
Find a path to point A.              $ <--- Printed text.

"Find a path to point A.             $ Returned STRING.
""

    format(                          $ Create a STRING,
    "The sum of ^1 and ^2 is ^3.     $   substituting
    ""                               $   elements in the
    [100 300 400]                    $   LIST for marked
    )                                $   locations.
```

```
"The sum of 100 and 300 is 400.      $ Returned STRING.
""

    define(                                $ Use of these functions
     {name -> CREATE_ENTRY                 $   within a user-defined
      member_of -> function                $   function.
      arguments -> []
      temporary -> [a b]
      algorithm ->
          [output(
           "Please enter name and age.
           "Name: "
           )
          set(a release(input() -1))       $ Drop c.r. char.
          output("Age: ")
          set(b parse())                   $ Enter age (NUMBER).
          output(
           format(
            "
            "Creating a new entry for
            "^1, age ^2.
            ""
            [.a .b]                        $ a=STRING, b=NUMBER.
           )
          )
         ...
         ]
     }
    )

{name -> CREATE_ENTRY
 member_of -> function
 ...
 }

    create(a variable)          $ Define temporary
                                $   variables.
{name -> A
 member_of -> variable
 }

    create(b variable)

{name -> B
 member_of -> variable
 }
```

```
    create_entry()
Please enter name and age.
Name: Joseph Smith
Age: 34

Creating a new entry for
Joseph Smith, age 34.

{name -> NIL
 member_of -> element
 }
```

Note that the functions "parse", "display", "input" and "output" all operate using the left margin as their reference. There is no automatic 5-space indentation for input as is the case when the user enters UNITs during the normal execution cycle.

Also, note that the "format" function is not an output function, but merely constructs a STRING which may be used as an argument to "output". Each occurrence of the sequence "^N", where N is a positive integer value, occurring in the STRING used as the first argument to "format", is replaced by the displayed form of the UNIT appearing as the N'th element in the LIST given as the second argument to "format". The UNITs to be inserted are evaluated prior to use in the resultant STRING. In the context of this LIST, STRINGs are treated in a special manner, in that they are not converted to their displayed form (including the surrounding double-quotes) but, rather, simply merged into the format STRING. This is the case for the name "Joseph Smith" in the above example: the STRING for this name appears without double-quotes in the output text.

Communication with files is accomplished by six built-in functions in STAR: "save", "stash" and "load" for transfer of UNITs, and "read", "write" and "extend" for transfer of text. The use of these functions is illustrated in Example 20.

**Example 20:**

```
define(                          $ Define a function
 {name -> SQUARE                 $   for squaring NUMBERs.
  member_of -> function
  arguments -> [a]
  algorithm -> [return(*(.a .a))]
  }
 )

{name -> SQUARE
 member_of -> function
 arguments -> [a]
 algorithm -> [return(*(.a .a))]
 }
```

```
    create(a variable)              $ Define variable "a".

{name -> A
 member_of -> variable
 }

    save("file1")                   $ Save all alterations to
                                    $   the knowledge base in
"file1"                             $   the file "file1".

    revise(square [])               $ This is a method for
                                    $   removing a named
{}                                  $   RECORD from STAR.

    revise(a [])

{}

    load("file1")                   $ Load definitions for
                                    $   "square" and "a" from
"file1"                             $   file.

    square(16)

256

    stash(                          $ "stash" is similar to
     "file2"                        $   "save" except that it
     [a]                            $   only saves the named
     )                              $   RECORDs provided in
                                    $   the LIST as its
"file2"                             $   second argument.

    stash(
     "file3"
     [square]
     )

"file3"

    write(                          $ Save text to file.
     "text1"
     "Haydn, Mozart and Beethoven were the
     "central composers of the Classical period.
     ""
     )
```

```
"text1"

    read("text1")                    $ Read text from file.

"Haydn, Mozart and Beethoven were the
"central composers of the Classical period.
""

    extend(                          $ Append text to file.
     "text1"
     "These composers contributed a great
     "influence to later composers in the Romantic
     "and more recent periods.
     ""
     )

"text1"

    read("text1")

"Haydn, Mozart and Beethoven were the
"central composers of the Classical period.
"These composers contributed a great
"influence to later composers in the Romantic
"and more recent periods.
""
```

The functions "save" and "stash" store the definitions of named RECORDs in a textual format, so that these definitions may be examined or altered in a general-purpose text editor. The definitions for named RECORDs to be stored are given in the form of "define" EXPRESSIONs which, when reloaded and evaluated, create the definitions for the named RECORDs in question. The function "load" completes the cycle, reading a file of UNITs in textual format and evaluating each UNIT as it is parsed.

Due to the nature of the "load" function, it can be used in several ways. In combination with the "save" and "stash" functions, it can be used to reconstruct definitions for named RECORDs which were saved in an earlier session with STAR. As well, the "load" function can be used to load in definitions of named RECORDs which are created using a general-purpose text editor of the programmer's choice. Finally, the "load" function can be used to execute arbitrary stored commands to the STAR interpreter, as the UNITs to be loaded are not limited to "define" EXPRESSIONs for named RECORDs.

Since the "load" function involves the evaluation of UNITs specified in a file, it is conceivable that the file to be loaded may contain within itself a call to "load". When evaluated, this results in a nested call to load yet another file. Such operations are quite acceptable and may be advantageous in certain cases.

The functions "read", "write" and "extend" form the textual interface with files. These routines are of a general nature, transferring STRINGs comprising whole files or major portions of files. At some point, the programmer may wish to take advantage of some of the more flexible capabilities for file input/output (and terminal input/output) which are provided by general-purpose languages like C. If this is the case, it is recommended that the programmer set up external functions in STAR (see Section 16) which interface directly with the desired input/output functions in the general-purpose language.

Two other functions related to input and output have been introduced previously in Section 3, the functions "spell" and "unspell". Since UNITs are easily converted to STRINGs representing UNITs and vice versa, the built-in functions for input and output of UNITs and those for input and output of STRINGs can be used somewhat interchangeably. A further function, "scan", is also available which operates exactly like "unspell" except that syntax errors are not announced as with "unspell"; rather, the value "nil" is simply returned when a syntax error is detected. The "scan" function is useful in cases where the user of a program has entered a line of text, for example, which may or may not conform to the syntax of a UNIT. The input text may be gathered in the form of a STRING using the "input" function, and the "scan" function may then be used to test if the characters actually do describe a UNIT without risking the possibility of generating a STAR error message and subsequent error diagnostics and trace listings. (For example, the EXPRESSION "set(b parse())" in the definition of "create_entry", Example 19 above, could be replaced with a safer operation, "set(b scan(input()))".)

Lastly, there are two additional STAR built-in functions which fall roughly into the category of input/output functions and are thus discussed at this point. These are the functions "pause" and "system". The "pause" function prints its argument, a STRING, at the user terminal in a manner similar to the function "output" and proceeds by taking a UNIT entered by the user (similar to the "parse" function), evaluating that UNIT (similar to "evaluate") and displaying the result (similar to "display"); "pause" repeats this process indefinitely until the value "nil" is entered as input. When a call to "pause" is inserted in the definition of a user-defined function, it has the effect of suspending the execution of that function and invoking an environment similar to an additional level of the STAR interpreter's main evaluation loop, with the difference that input UNITs are not indented the traditional five spaces but appear at the left margin, and entering the value "nil" as mentioned above will cause an end to the process. The "pause" function is often useful in setting break points within user-defined functions for debugging purposes.

The "system" command provides an interface to the operating system of the host computer (this function may work somewhat differently or not at all on operating systems other than UNIX, depending upon the particular operating system). The "system" command takes a single argument, a STRING specifying an arbitrary command to the operating system, and proceeds by executing that command at the operating system level. The character sequence contained within the STRING must therefore meet the syntax requirements for commands in the host operating system language. The "system" command can be used to invoke text editors, send mail, list directories, call executable files and perform other related tasks. These operations may be initiated by the programmer during a session with the STAR interpreter, or may be initiated in the course of operating an application system.

## 13. Logical Operations

The logical values in STAR are the two named RECORDs "true" and "false", contained in the class "boolean" which is a subclass of "element". The implementation of logical values as named RECORDs allows for efficient operation on these values, as comparisons may be made simply on the basis of their memory addresses.

In addition, STAR includes a special named RECORD "nil", which is used in a general sense to indicate the absence of a meaningful value. Thus, logical decisions in STAR may be made on the basis of "true versus false" or on the basis of "nil versus non-nil". Both are acceptable.

As a matter of choice, the built-in functions in STAR which relate to logical operations are biased towards the "true versus false" scheme. To facilitate this, the function "null" is included, which takes a single UNIT as its argument and returns "true" if the UNIT is "nil" and "false" otherwise. Logical operations involving the "nil versus non-nil" way of thinking are performed using EXPRESSIONs of the form "null(.x)" and "~ null(.x)" (abbreviation for "not(null(.x))") along with the built-in boolean functions.

The primary logical functions of STAR are "not", "and" and "or", abbreviated using the characters tilde ("~ "), ampersand ("&") and vertical bar ("|"), respectively. In addition, there are several built-in functions in STAR which take non-boolean arguments and return boolean values as results. These functions include "less" and "greater" (abbreviated "<" and ">"), used in comparing NUMBERs, the function "equal" (abbreviated "="), used in comparing two arbitrary UNITs, and individual predicates for each type of UNIT ("number", "token", "string", "list", "record", "expression" and "connection"). All of these functions appear in the example below.

**Example 21:**

```
    or(                          $ The functions "not",
     not(and(true false))        $   "and" and "or".
     false
     )

{name -> TRUE
 member_of -> boolean
 }

    |(~ &(true false) false)     $ Abbreviated format.

{name -> TRUE
 member_of -> boolean
 }

    &(~ null(box3)  ~ null(hand7))    $ Use in conjunction
                                      $   with "null".
```

```
{name -> TRUE
 member_of -> boolean
 }

    less(3 2)                    $ "Less than".

{name -> FALSE
 member_of -> boolean
 }

    <(3 2)                       $ Abbreviated format.

{name -> FALSE
 member_of -> boolean
 }

    greater(3 2)                 $ "Greater than".

{name -> TRUE
 member_of -> boolean
 }

    >(3 2)                       $ Abbreviated format.

{name -> TRUE
 member_of -> boolean
 }

    equal(23 46)                 $ Equality of UNITs.

{name -> FALSE
 member_of -> boolean
 }

    =(ADD ADD)

{name -> TRUE
 member_of -> boolean
 }

    =(
     [1 2 3]
     [8 2 3]
     )
```

```
{name -> FALSE
 member_of -> boolean
 }

    number(SELECT)          $ Return "true" if the
                            $   UNIT is a NUMBER,
{name -> FALSE
 member_of -> boolean
 }

    token(SELECT)           $ ...a TOKEN,

{name -> TRUE
 member_of -> boolean
 }

    record(subtract)        $ ...a RECORD.

{name -> TRUE
 member_of -> boolean
 }
```

The function "equal" works as follows. In most cases, equality is measured in terms of similarity in the displayed forms for two UNITs. This is the case with NUMBERs, TOKENs, STRINGs, LISTs, unnamed RECORDs and EXPRESSIONs. For named RECORDs and CONNECTIONs, however, comparison is made on the basis of memory address. This is a shortcut in the case of named RECORDs, as the STAR interpreter does not allow two RECORDs to share the same name. In the case of CONNECTIONs, this allows for differentiation between distinct routines and data structures even if their labels for display purposes happen to be the same.

A further set of seven built-in functions provides more extensive comparison and evaluation capabilities which result in the generation of boolean values. These are the functions "in", "subset", "isa", "within", "exists", "every" and "which", illustrated in Example 22, below.

**Example 22:**

```
    in(                     $ Set membership of a UNIT
     add                    $   in a LIST of UNITs.
     [add subtract multiply divide]
     )

{name -> TRUE
 member_of -> boolean
 }
```

```
   subset(                  $ Subset relation between
    [3 4 7]                 $   one LIST and another
    [1 4 7 5 3]             $   LIST.
    )

{name -> TRUE
 member_of -> boolean
 }

   isa(                     $ Tests if a named RECORD
    add                     $   is a member (directly
    attribute               $   or indirectly) of a
    )                       $   particular class.

{name -> FALSE
 member_of -> boolean
 }

   isa(add concept)         $ All named RECORDs are
                           $   members of the class
{name -> TRUE               $   "concept".
 member_of -> boolean
 }

   within(                  $ Tests if a class is a
    function                $   subclass (directly or
    attribute               $   indirectly) of another
    )                       $   class.

{name -> FALSE
 member_of -> boolean
 }

   create(a variable)       $ Variable "a", used below.

{name -> A
 member_of -> variable
 }

   exists(                  $ Existential quantification
    [1 4 2 3 8]             $   on a LIST of UNITs. (Does
    a                       $   there exist an element of
    '>(.a 6)                $   the LIST "[1 4 2 3 8]"
    )                       $   which is greater than 6?)
```

```
{name -> TRUE
 member_of -> boolean
 }

    every(                    $ Universal quantification.
     :function                $ (":" abbreviates the
     a                        $   function "enumerate".)
     '~ null(get(.a abbreviation))
     )

{name -> FALSE
 member_of -> boolean
 }

    which(                    $ Return a subset containing
     :function                $   only elements for which
     a                        $   the EXPRESSION is true.
     '~ null(get(.a abbreviation))
     )

[negate add subtract multiply divide dot enumerate
 equal less greater not and or quote]
```

The functions "in" and "subset" operate primarily on individual UNITs versus LISTs of UNITs, although it is also possible to use them in the context of named RECORDs versus LISTs of the enumerated members of classes. The functions "isa" and "within", on the other hand, operate exclusively on named RECORDs and classes.

The functions "exists", "every" and "which" offer a powerful general capability for search operations applied to the elements of a LIST. As is the case with the functions "in" and "subset", these functions can be used either in the context of LISTs of arbitrary UNITs or in the context of LISTs of the enumerated members of classes (as in the last two entries in the example above).

## 14. Programming Constructs

### Evaluation

Some of the basic functions of STAR related to evaluation have already been introduced. The "quote" function spares a UNIT from evaluation and is especially useful in preserving EXPRESSIONs in their unevaluated form for programming purposes. The function "prepare" forces evaluation of the top level of elements within a LIST or RECORD, as these elements are unaffected by evaluation of the entire LIST or RECORD. The function "apply" takes a function and a LIST of arguments and causes an EXPRESSION involving these quantities to be evaluated.

The other related function which has not yet been introduced is the general function "evaluate". This function causes an additional iteration of evaluation to occur for the UNIT supplied as its argument. Example 23, below, illustrates the "evaluate" function in the context of some of its common uses.

**Example 23:**

```
    '*(40 5)                    $ An EXPRESSION initially
                                $   quoted upon entry.
*(40 5)

    evaluate(#)                 $ Subsequent evaluation of
                                $   the EXPRESSION.
200

    [+(2 3) +(4 5)]             $ A LIST with EXPRESSIONs
                                $   as elements.
[+(2 3) +(4 5)]

    evaluate(select(# 1))       $ Evaluation of one of the
                                $   elements.
5

    ''&(true false)             $ A redundantly quoted
                                $   EXPRESSION.
'&(true false)

    evaluate(#)                 $ Successive evaluation of
                                $   the EXPRESSION.
&(true false)

    evaluate(#)

{name -> FALSE
 member_of -> boolean
 }
```

### Simple versus compound statements

Related to the conventional programming structures one might find in a language like C or PASCAL, STAR contains a full set of conditional evaluation functions, iterative looping functions with optional explicit control-flow directives, and a general capability for substituting a "compound statement" in the context of evaluating a single UNIT. The mechanism for compound statements is covered first.

The "do" function in STAR takes a single LIST as its argument and sequentially evaluates the elements of the LIST, optionally returning "nil" or an explicitly-specified UNIT as its result. The operation of the "do" function is similar in a number of ways to the execution of user-defined functions in STAR, minus the apparatus for arguments and temporary variables. For user-defined functions, the "return" function may be used to cause an immediate exit from the execution process, with a specified value returned by the function. In the context of "do", the "result" function provides this capability.

The "result" function works in the same manner "return" does: it forms a single-entry RECORD with attribute "result_value" associated with the argument to "result", and returns this UNIT. The "do" function then recognizes this special construction and acts accordingly by halting its execution and returning the extracted value. The following example illustrates the use of both "do" and "result".

**Example 24:**

```
    $ Assume previously defined: variables "a" and
    $   "b", with at least one binding each.

    do(                          $ Sequentially evaluate the
     [set(a 10)                  $   elements of the given
      set(b 20)                  $   LIST.
      display(+(.a .b))
      ])
30                               $ <--- Displayed value.

{name -> NIL                     $ Returned value is "nil".
 member_of -> element
 }

    do(                          $ This version returns the
     [set(a parse())             $   displayed sum as its
      set(b parse())             $   result. Note that the
      output(                    $   EXPRESSION "display(555)"
       format(                   $   is never evaluated.
        "The sum is ^1.
        ""
        [+(.a .b)]
        ))
      result(+(.a .b))
      display(555)
      ])
10                               $ <--- Input for "a".
20                               $ <--- Input for "b".
The sum is 30.                   $ <--- Output.
```

```
30                          $ Return value specified by
                            $   "result".

    result(30)              $ Operation of the "result"
                            $   function in isolation.
{result_value->30}
```

Two "control-flow" functions have been introduced thus far, "return" and "result". There are also three others, "break", "skip" and "stop", which all work in the same manner, constructing special RECORDs which are recognized at higher levels as flags indicating the various actions to be taken. The implementation of these functions in this manner allows a number of interesting possibilities, as for example, the occurrence of "result(result(...))" within two levels of "do", causing an exit simultaneously from both. (An instance of this type of "multiple-level control flow" operation is given in Example 27, later in this section.)

### Conditional evaluation functions

STAR provides three functions for conditional evaluation, the "if" function for simple conditionals, "ifelse" for two-way conditionals, and "branch" for flexible multi-way conditionals. The operation of these functions is illustrated in the example below. The "branch" function uses a special structure for its second argument: a LIST of RECORDs, each containing the attributes "case" and "action".

**Example 25:**

```
    .x                      $ Assume "x" has the
                            $   value "-20".
-20

    if( |(<(.x 0) >(.x 100))    $ Use of the "if"
     'output(                   $   function. No
       "Unacceptable value.     $   action taken if
       ""                       $   the first argument
       )                        $   is "false".
     )
Unacceptable value.

"Unacceptable value.
""

    .y                      $ Assume "y" has the
                            $   value "75".
75
```

```
    ifelse( |(<(.y 0) >(.y 100))      $ Use of the "ifelse"
     'output(                         $   function.  If the
       "Unacceptable value.           $   first argument
       "")                            $   is "true", return
     'output(                         $   the result of
       format(                        $   evaluating the
        "^1 percent.                  $   second argument.
        ""                            $   Otherwise, return
        [.y]))                        $   the result of
     )                                $   evaluating the
75 percent.                           $   third argument.

"75 percent.
""
```

```
    output(                           $ Alternate form for
     ifelse( >(.y 100)                $   the previous
      '"Unacceptable value.           $   example.  The
       ""                             $   "ifelse" function
      'format(                        $   returns a STRING
       "^1 percent.                   $   to be used by
       ""                             $   the "output"
       [.y])                          $   function.
    ))
75 percent.

"75 percent.
""
```

```
    if( |(<(.x 0) >(.x 100))          $ The "do" function is
     'do([                            $   used within an
        output(                       $   "if" or "ifelse"
         "Unacceptable value.         $   EXPRESSION if more
         "Enter a new value: ")       $   than one action is
        set(x scan(input()))          $   to be taken.  Note
        result(.x)                    $   that the result of
        ]))                           $   "do" is returned
Unacceptable value.                   $   as well by the
Enter a new value: 64                 $   "if" function.

64
```

```
output(                                  $ Application of
 format(                                 $  "branch". Let
  "The student's ^1 performance.         $  ".y"="75" as
  ""                                     $  above. Assume
  [                                      $  "grade(.y)"
   branch( grade(.y)                     $  produces a
    [                                    $  TOKEN "A",
     {case->A action->"superior"}        $  "B", "C", "D"
     {case->B action->"good"}            $  or "F". This
     {case->C action->"average"}         $  TOKEN is then
     {case->D action->"poor"}            $  compared with
     {case->F action->"failing"}         $  the evaluated
    ])                                   $  "case" UNITs,
  ]))                                    $  & if matched,
The student's poor performance.          $  the specified
                                         $  "action" UNIT
"The student's poor performance.         $  is evaluated
""                                       $  and returned.
```

```
  .z                                     $ Assume "z" has the
                                         $  value "127".
127
```

```
  branch( true                           $ Alternatively, the UNIT
   [                                     $  to be matched may be
    {case -> <(.z 0)                     $  a constant such as
     action -> .set(z 0)}                $  "true", causing a
    {case -> >(.z 100)                   $  search for the first
     action -> .set(z 100)}              $  evaluated "case" UNIT
    {case -> true                        $  which matches this
     action -> .z}                       $  value.
   ])

100
```

```
  branch( maximum(.x .y)                 $ Finally, "branch" may
   [                                     $  be used in a manner
    {case -> minimum(.x .y)              $  where both the
     action -> nil}                      $  evaluation of the
    {case -> .x                          $  UNIT to be matched
     action -> set(y .x)}                $  and evaluation of the
    {case -> .y                          $  "case" UNITs have
     action -> set(x .y)}                $  an effect on the
   ]                                     $  action taken. Here,
  )                                      $  the variable of
                                         $  lesser value is
```

```
{name -> X                          $  assigned the value of
 member_of -> variable              $  the other variable.
 bindings -> [75]
 }
```

It can be seen from the above example that there are a number of different ways to use the functions "if", "ifelse" and "branch". In some cases, it is the conditional evaluation of UNITs that is of primary importance. In other cases, it is the values returned by these functions. In still other cases, it is both.

The "branch" function has an added flexibility in that both the UNIT to be matched and the "case" UNITs are evaluated. Thus, the match may compare the result of evaluating an EXPRESSION with several constants, or it may match a constant against the results of evaluating several EXPRESSIONs, or it may match the evaluation of one EXPRESSION against the evaluation of other EXPRESSIONs. Each of these cases is illustrated above.

### Looping Constructs

There are four iterative looping functions provided in STAR. These are "repeat", "while", "through" and "for". All of these functions recognize special control directives generated by the functions "break" (exit the loop, returning a value) and "skip" (advance to the next iteration).

The looping functions in STAR operate on LISTs of UNITs to be evaluated, similar to both the execution of user-defined functions and the operation of the "do" function. Analogously, the "break" and "skip" functions behave in a similar manner to "return" for user-defined functions and "result" for the "do" function. The following example illustrates the use of looping functions within STAR, with and without the additional use of "break" and "skip".

### Example 26:

```
$ Assume previously defined: "block", "cylinder",
$   "dimensions", "color", "supported_by", "a",
$   "o", "min_pos_val", "i", "j", "block_1",
$   "block_14", "cylinder_7", "block_3",
$   "block_44", "block_8", "cylinder_5",
$   "block_11" and "block_2".
```

```
   repeat(                          $ Use of "repeat". The
    [output(                        $  LIST of UNITs is
      "Enter a number               $  repeatedly processed,
      "between 1 and 10.            $  one UNIT at a time,
      "> ")                         $  with the "repeat"
     set(a parse())                 $  function monitoring
     if(                            $  the evaluation results
      ~ |(<(.a 1) >(.a 10))         $  in search of control
      'break(.a))                   $  directives from "skip"
     ])                             $  or "break". Here, the
Enter a number                      $  "break" function is
between 1 and 10.                   $  used to indicate an
> 144                               $  exit from the loop
Enter a number                      $  when a value in the
between 1 and 10.                   $  requested range has
> -26                               $  been entered. (The
Enter a number                      $  operation of "break"
between 1 and 10.                   $  in isolation is
> 7                                 $  illustrated at end of
                                    $  this example.)
7

   .o                               $ Suppose that variable
                                    $  "o" (current object
{name -> BLOCK_1                    $  under consideration)
 member_of -> block                 $  has "block_1" as its
 dimensions -> [3 4 2]              $  value.
 color -> red
 supported_by -> block_14
 }

   while(                           $ Use of "while" to find
    '~ =(                           $  the base of the column
      get(.o supported_by)          $  of objects supporting
      table                         $  the given object (this
      )                             $  may be the object
    [set( o                         $  itself). The first
      get(.o supported_by))         $  argument to "while"
     ])                             $  must evaluate to
                                    $  "true" for each cycle.
{name -> NIL
 member_of -> element
 }
```

```
    .o                          $ The resultant value of
                                $  "o" might be, for
{name -> CYLINDER_7             $  example, the object
 member_of -> cylinder          $  "cylinder_7".
 dimensions -> [6 3]
 color -> orange
 supported_by -> table
 }

    .min_pos_val                $ Suppose "min_pos_val" is
                                $  initialized to "1000".
1000

    through(                    $ "through" binds a
     [7 1 3 -2 4 8 9 6]         $  variable sequentially
     i                          $  to each element of a
     [if(<(.i 0) 'skip())       $  LIST (1st argument) as
      if(                       $  it evaluates elements
       <(.i .min_pos_val)       $  of a second LIST (3rd
       'set(min_pos_val .i))    $  argument). Here, the
      ])                        $  minimum positive value
                                $  in a LIST is found.
{name -> NIL
 member_of -> element
 }

    .min_pos_val                $ Resultant value for
                                $  "min_pos_val".
1

    through( :block o           $ The "enumerate"
     [if(                       $  function (":" is
       =(get(.o supported_by) nil)  $  the abbreviation
       'do(                     $  for "enumerate")
        [output(                $  may be used to
         format(                $  provide a LIST
          "Support for ^1?      $  of the members
          "> "                  $  of a class, so
          [get(.o name)]))      $  that "through"
        modify( .o supported_by $  may perform some
         locate(parse()))       $  operation on
        ]))                     $  each of these
     ])                         $  named RECORDs.
Support for BLOCK_3?            $  Here, values for
> BLOCK_44                      $  the attribute
Support for BLOCK_8?            $  "supported_by"
> CYLINDER_5                    $  are requested
```

```
Support for BLOCK_11?                $ for members of
> BLOCK_2                            $ "block" without
                                     $ such values.

{name -> NIL
 member_of -> element
 }

    for(                             $ The "for" function
     'do([set(i 0) set(j 1)])        $  begins by evaluating
     '~ >(.i 5)                      $  its first argument,
     'do(                            $  then cycles through
       [set(i +(.i 1))               $  its LIST of UNITs,
        set(j *(.j 3))])             $  evaluating its second
     [output(                        $  argument prior to
       format(                       $  each cycle and its
        "3**^1 = ^2                  $  third argument
        ""                           $  following each cycle.
        [.i .j]))                    $  If the second
      ])                             $  argument does not
3**0 = 1                             $  evaluate to "true"
3**1 = 3                             $  for a given cycle,
3**2 = 9                             $  "for" halts its
3**3 = 27                            $  looping process
3**4 = 81                            $  and returns "nil".
3**5 = 243

{name -> NIL
 member_of -> element
 }

    break(44)                        $ Operation of "break" in
                                     $  isolation.
{break_value->44}

    skip()                           $ Operation of "skip" in
                                     $  isolation.
{skip_value->nil}
```

The "skip" and "break" functions may be used with all four functions, "repeat", "while", "through" and "for". When using "skip" or "break", it is important to recall that their operation works indirectly by setting up special-purpose values to be detected by the looping functions in order to produce a skip to the next iteration or an exit from the looping process. For this reason, the programmer must be careful to ensure that the special value returned by "skip" or "break" does in fact propagate back to the level of the target looping function. As an example, the "if" function in the above illustration for "repeat" performs this task by conveying the special value returned by "break" back to

the level where it is detected by "repeat".

As mentioned previously in this section, a side-effect of the mechanisms for control-flow in STAR is that it is possible to perform multiple control-flow operations from a single vantage point. The example below provides a simple example of this sort. The "through" EXPRESSION given performs a set union operation, appending to the LIST given as the value of "a" all elements not appearing in this LIST but appearing in the LIST given as the value of "b". (The following is provided only for the sake of example as there is a built-in function "union" included in STAR.)

**Example 27:**

```
$ Assume previously defined: variables "a", "b",
$  "aa" and "bb".

.a                  $ Suppose that the value of "a" is
                    $   as follows.
[2 5 3 7 1]

.b                  $ Suppose that the value of "b" is
                    $   as follows.
[4 8 7 0 2 6]

through(.b bb                          $ Set union.
 [through(.a aa                        $   Note the
   [if(==(.bb .aa) 'break(skip()))     $   combined use
   ])                                  $   of "break"
  set(a insert(.a -1 .bb))             $   and "skip".
  ])                                   $   (Explanation
                                       $   follows.)
{name -> NIL
 member_of -> element
 }

.a                  $ Resultant value for "a".

[2 5 3 7 1 4 8 0 6]

break(skip())                          $ "break" and "skip" in
                                       $   isolation.
{break_value->{skip_value->nil}}
```

In this example, the outer loop cycles through the elements of the LIST given for "b". For each element in this LIST, a search is conducted through the elements of LIST given for "a". If the element is found, an exit from the inner loop is made, and, additionally, a skip to the next iteration of the outer loop is made. In this manner, the LIST for "a" is appended only with those elements of the LIST for "b" which are not found in the LIST

for "a".

It may be informative to trace the return values involved in the operation of the above "through" EXPRESSION. The combination of "break" and "skip" produces a nested RECORD value as listed at the end of the example. The "if" function returns this RECORD to the inner "through" function, which detects the "break_value" attribute signaling an exit from its looping process. The value of "break_value" in this RECORD is then returned by the inner "through" function to the outer "through" function, which detects the "skip_value" attribute as signaling a skip to its next iteration.

In general the functions "return", "result", "break", "skip" and "stop" (to be described in the next section) may be used in combination in a number of different ways. Despite the versatility allowed by this setup, however, it is also possible through carelessness to define programming structures whose operation is subtly different from that intended by the programmer. It is therefore advised to use these functions in combination only where the effect of their operation is clearly understandable.

## 15. Rule-based Operation

Rule-based operation in STAR is similar in many ways to the application of functions to sets of arguments. The structures analogous to functions are classes of rules, as defined in the class/subclass hierarchy of the STAR knowledge base.

As one might apply a function to a number of UNITs provided as arguments, in a similar manner one may invoke a class of rules, passing a number of UNITs to be used in a sense equivalent to that of arguments to a function. Upon completion of the operation resulting from the invocation of a class of rules, a value is returned similar to the return value from a function.

The actual operations involved in processing classes of rules are of course quite different from those involved in processing functions. In STAR, the "rules" are production rules and are processed in the manner of executing a production system. The operation in detail is described in the remainder of this section.

The example below has been provided at this point in order to serve as a focus for the following discussion. As has been the case for other examples provided above, the named RECORDs used within the example are assumed to have been defined previously and the definitions are simply retrieved at this point. If the reader wishes to enter the definitions for these named RECORDs in order to view the action of the example from within STAR, it is simply required to enclose each defining RECORD within a call to the function "define".

The example below involves a simple parsing operation, using STAR EXPRESSIONs involving the constants "true" and "false" and the functions "and","or" and "not" (in abbreviated forms) as a basis for the range of acceptable inputs. Input is provided in the form of a STAR STRING. For example, the parsing operation should succeed given the following STRINGs.

"&(~ true false)" "~ ~ true"

"false" "|(&(false true) true)"

The parsing operation should not succeed, however, given the following STRINGs.

"true false" "|(false)"

"&(true perhaps)" "~ ()false|("

Only brief comments appear for the entries below as the definitions involved are described in greater detail in the context of the ensuing discussion.

**Example 28:**

```
    $ Assume previously defined: "ruleset_1", "given",
    $   "map_to", "a", "b", "c", "x", "r_1", "r_2",
    $   "r_3", "r_4", "r_5" and "r_6".

    a               $ Argument to ruleset_1. Stores current
                    $   state of STRING being parsed.
{name -> A
 member_of -> variable
 bindings -> []
 }

    b               $ Used by "given" & "map_to" to pass the
                    $   location of a substring in the STRING.
{name -> B
 member_of -> variable
 bindings -> []
 }

    c               $ Used by "given" & "map_to" to pass the
                    $   length of a substring.

{name -> C
 member_of -> variable
 bindings -> []
 }

    x               $ Argument to "given" & "map_to".

{name -> X
 member_of -> variable
 bindings -> []
 }
```

```
   given          $ Returns "true" if substring found within
                  $   the STRING to be parsed.
{name -> GIVEN
 member_of -> function
 arguments -> [x]
 algorithm ->
     [set(b find(.x .a))
      set(c length(.x))
      return(<(.b length(.a)))
     ]
}

   map_to         $ Replaces the current substring with a
                  $   new substring (argument to "map_to").
{name -> MAP_TO
 member_of -> function
 arguments -> [x]
 algorithm ->
     [set(a
       join(
        join(fetch(.a .b) .x)
        release(.a +(.b .c))
        ))
     ]
}

   ruleset_1              $ Definition of the class of rules
                          $   for the parsing operation.
{name -> RULESET_1
 member_of -> class
 subclass_of -> rule
 members -> [r_1 r_2 r_3 r_4 r_5 r_6]
 arguments -> [a]
 temporary -> [b c]
}

   r_1            $ Rule 1.  If "true" found, replace it
                  $   with the character "$".
{name -> R_1
 member_of -> ruleset_1
 mode -> multiple_application
 condition -> given("true")
 action -> map_to("$")
}
```

```
   r_2              $ Rule 2. If "false" found, replace it
                    $   with the character "$".
{name -> R_2
 member_of -> ruleset_1
 mode -> multiple_application
 condition -> given("false")
 action -> map_to("$")
 }

   r_3              $ Rule 3. If an application of "and" is
                    $   found, replace with the character "$".
{name -> R_3
 member_of -> ruleset_1
 mode -> multiple_application
 condition -> given("&($ $)")
 action -> map_to("$")
 }

   r_4              $ Rule 4. If an application of "or" is
                    $   found, replace with the character "$".
{name -> R_4
 member_of -> ruleset_1
 mode -> multiple_application
 condition -> given("|($ $)")
 action -> map_to("$")
 }

   r_5              $ Rule 5. If an application of "not" is
                    $   found, replace with the character "$".
{name -> R_5
 member_of -> ruleset_1
 mode -> multiple_application
 condition -> given("~ $")
 action -> map_to("$")
 }

    r_6             $ Rule 6. If current STRING contains only
                    $   "$", return "true" from invocation.
{name -> R_6
 member_of -> ruleset_1
 mode -> single_application
 condition -> =(.a "$")
 action -> stop(true)
 }

   invoke(ruleset_1 ["&(true |(true false))"])
```

```
{name -> TRUE
 member_of -> boolean
 }

    invoke(ruleset_1 ["~ ~ &(false false)"])

{name -> TRUE
 member_of -> boolean
 }

    invoke(ruleset_1 ["|(true ~ )"])

{name -> NIL
 member_of -> element
 }

    invoke(ruleset_1 ["&(&(true true))"])

{name -> NIL
 member_of -> element
 }
```

It can be seen from the above example that a number of different components combine in the operation of rules in STAR. The functions "given" and "map_to", above, aid in the operation of the rules by carrying out tasks which are common to many rules. The function "given" detects when a specified substring appears within the current state of the STRING being parsed, and sets the global variables "b" and "c" to convey the boundaries of the substring to the function "map_to". This latter function replaces the specified substring with a new substring provided as its argument. Using these functions, each rule is given the task of detecting a particular acceptable sequence and replacing that sequence with the symbol "$", allowing higher-level sequences to be detected. In this manner, an acceptable STRING is gradually reduced, having portions of itself replaced by "$" characters until only a single "$" remains as the entire STRING. For an unacceptable STRING, the rules will be unable to complete this reduction and at some point the process will fail, causing "nil" to be returned. Before examining the operation of the above example in detail, however, it is necessary to consider the mechanisms by which rule-based operation is carried out in STAR.

The definition for a class of rules can be seen from the above entry for "ruleset_1" to comprise elements found in both the definition of classes and of functions. The set of arguments and temporary variables is treated in the same manner as in the operation of functions. When a class of rules is invoked, the LIST appearing as the second argument to "invoke" is split up into individual UNITs which become new bindings of the arguments to the class of rules for the duration of the invocation (the "invoke" function thus parallels the syntax of "apply"). Thus, in the calls to "invoke", above, the STRINGs in the LISTs sent to "invoke" become new bindings for the variable "a". As well, variables listed in the "temporary" entry for the class of rules receive new bindings initialized to

"nil". This occurs for the variables "b" and "c" during the applications of "invoke" in the above example. The new bindings for both arguments and temporary variables are then removed as the invocation completes.

The "invoke" function makes use of two built-in STAR variables, "control" and "alternatives", in carrying out its designated operation. These variables belong to the class "variable", and are specially set aside for use with "invoke". When "invoke" is called, "control" is given a new binding to the class of rules being applied ("ruleset_1" in the above example). The variable "alternatives" is also given a new binding which is a LIST containing all direct or indirect members of that class of rules ("r_1", "r_2", "r_3", "r_4", "r_5" and "r_6" in the above example).

Each rule has entries for the attributes "condition" and "action", in addition to other information. The UNITs associated with these attributes are referred to, respectively, as the condition and the action of a rule. The operation of "invoke" is to scan through the LIST bound to "alternatives", looking for the first rule whose condition evaluates to "true". When such a rule is found, the action of that rule is evaluated (the rule "fires") and "invoke" returns to the top of the LIST for a subsequent pass. This scanning process continues until one of three things happens: (1) no "condition" UNIT evaluates to "true" on a given pass, (2) the "stop" function is used to explicitly cause an exit from "invoke", or (3) the LIST becomes empty (for reasons described later).

The first two causes for an exit from "invoke" are illustrated in the above example. When an unacceptable STRING is parsed, there is some point, possibly on the first pass or possibly on a subsequent pass, at which all six rules in the class "ruleset_1" fail to fire. The STRING has been reduced as far as possible and yet the successful end condition of a single "$" character remaining has not been reached (this would cause rule "r_6" to fire). In such a case, the operation of "invoke" ceases following testing of the last rule, and the UNIT "nil" is returned for the call to "invoke". In the case of an acceptable STRING, there is some point at which the STRING has been reduced to a single "$" character, at which time rule "r_6" fires, causing an exit from "invoke" by the second method listed above: use of the "stop" function. The "stop" function works in the same manner as do the functions "return", "result", "break" and "skip"; that is, it does not perform the exit itself, but rather returns a special RECORD value (for "stop" this RECORD contains the attribute "stop_value") to be detected by "invoke", thus indirectly causing the exit. The UNIT to be returned for the call to "invoke" is given as the argument to "stop" and appears as the value of the attribute "stop_value" in the RECORD returned by "stop".

The third mechanism by which a call to "invoke" may return involves the use of "rule modes". Each rule contains within its defining RECORD an entry for the attribute "mode" which specifies a value of either "single_test", "single_application" or "multiple_application". These three values are built-in named RECORDs in STAR, belonging to the class "rule_mode", a subclass of "element". The value specified for a particular rule determines whether or not it is to be removed from the "alternatives" LIST following the testing and/or firing operations. Conceivably, through the use of appropriate rule modes, the LIST of alternatives can be made to shrink in size until an exit is signaled by the absence of remaining rules in the LIST. In this case, as in the case of no tested rules firing, the value "nil" is returned. The following paragraphs describe the use of rule modes within STAR.

Rules with mode "single_test" are only considered once. That is, they are placed initially in the LIST bound to "alternatives", but following the first time "invoke" tests the conditions of such rules (and possibly evaluates their actions), they are removed from the LIST. Rules of this type have actions whose conditions may be true or false but are not expected to change during the course of a call to "invoke". Thus if the rule's condition evaluates to "false" on the first pass, it is expected to remain "false" for the remainder of the process and thus need not be reconsidered. If the condition evaluates to "true", the action is taken, and it is assumed that by taking this action, it need not be taken again. The above example contains no rules of this type, only rules of the remaining two types.

Rules with mode "single_application" and "multiple_application" generally appear more frequently than those with mode "single_test". A "single_application" rule may be tested an indefinite number of times as long as the condition evaluates to "false". On the first time the condition evaluates to "true", however, the rule is fired and also removed from the LIST of alternatives. Such rules are used when a condition is expected to change during rule-based operation, but the action, once completed, need never be repeated. In contrast, a "multiple_application" rule may be tested and fired as many times as required during the course of a call to "invoke".

In the above example, rule "r_6" is of type "single_application" while the remaining rules are of type "multiple_application". The designation of "single_application" for "r_6" is actually academic, as this rule causes a direct exit from the call to "invoke"; however, as the rule is indeed applied only once, it is useful to label it as such. The other rules are labeled "multiple_application" as the patterns they detect may occur more than once within the input STRING. For example, in order to parse the STRING "&(true true)", two applications of rule "r_1" must take place. Note that the actions of these rules effectively eliminate the conditions which cause them to fire by replacing the triggering substring with a new substring. This keeps the system from locking up in an infinite loop, with a rule constantly firing based on the same conditional evidence. In general, "multiple_application" rules will in some manner reset the conditions which cause them to fire, so that they do not fire again until a new instance of need arises.

At this point, it is possible to step through a sample parsing operation as performed in the above example. Consider the input STRING "&(true |(true false))". The following list describes the sequence of operations which takes place in the invocation of "ruleset_1" using this STRING as the argument.

**(1)** The call to "invoke" first creates new bindings for the variables "control" and "alternatives", set, respectively, to the class "ruleset_1" and a LIST containing "r_1" through "r_6".

**(2)** Next, new bindings are created for the variables "a", "b" and "c". The variable "a" is associated with the input STRING, while "b" and "c" are associated with the value "nil".

**(3)** The first pass through the LIST of alternatives results in rule "r_1" firing, causing the first occurrence of "true" in the input STRING to be replaced by "$".

(4) On the second pass, rule "r_1" fires again, replacing the second occurrence of "true" with "$". The value of "a" following this step is "&($ |($ false))".

(5) Next, rule "r_2" fires on the third pass, replacing the occurrence of "false" and resulting in a value of "&($ |($ $))" for "a".

(6) Rule "r_4" fires on the fourth pass, replacing the substring "|($ $)" with "$". Variable "a" then has the STRING "&($ $)" as its value.

(7) Rule "r_3" completes the reduction on the fifth pass, replacing the remaining characters with a single "$".

(8) On the sixth pass, then, rule "r_6" fires, having detected the completed parse. The "stop" function is called with an argument of "true", and detection of its returned RECORD causes an exit from the call to "invoke", with "true" returned as the result.

A few final comments may be said about rule-based operation in STAR before moving on to the integration of multilevel application systems. STAR is intended to provide a low-level yet flexible environment for the creation of AI application systems. Rule-based operation may be freely intermixed with algorithmic operation, with either component in control. Likewise, calls to "invoke" may be made from within the actions of rules, causing nested levels of rule-based operation to be set up ("control" and "alternatives" are given additional bindings, and a new class of rules is executed until its operation returns control to the former class). In cases where rule-based operation is nested to several levels, the "stop" function can be nested (e.g., "stop(stop(true))") to cause a simultaneous exit from more than one level.

On the other hand, it should be pointed out that STAR does not have built into its framework mechanisms for distinguishing between forward and backward chaining and for the generation of rule firing histories for the generation of explanations in reasoning. In cases where these are required, the programmer must use the basic mechanisms provided in STAR to define the necessary constructs. For the generation of explanations, a simple technique is to encode within the actions of all rules a call to some function which appends the necessary information to a global LIST stored within the semantic network of STAR. Subsequent retrieval of this LIST may be followed by the extracting of pseudo-English explanations from within the definitions of the rules themselves.

## Part IV: AI Application Systems

### 16. Design of Multilevel Systems

Applications in Artificial Intelligence often involve not only symbolic processing but extensive numerical processing as well, be it signal processing, quantitative modeling, linear programming or otherwise. A central focus in the design of STAR has been the facilitation of such hybrid or multilevel applications. As there exist a good number of traditional programming languages which perform numerical computation extremely well, STAR takes advantage of this by providing a flexible interface to both routines and data structures defined in these languages. Compiled code for routines and data structures defined in general-purpose languages may be linked together with the STAR intepreter, forming a unified operating environment in which both symbolic and numerical operations may be conducted in an integrated manner.

In some cases, the application may center around a set of previously existing analysis routines. In these cases, the STAR language may be used to add a controlling symbolic module which manages the execution of the routines, thus forming an AI application system without the need for reprogramming of the original routines. In other cases, matters of efficiency or portability may dictate the location of parts of an application system in general-purpose languages as opposed to the central symbolic language. In any case, the decision of which portions of an application system to implement symbolically and which to implement numerically is an important one and can affect the overall efficiency and effectiveness of the system to a large extent.

Once the decision has been made to implement certain portions of an application system in a general-purpose language (e.g., FORTRAN, C or PASCAL), the overall design can be carried out either in a top-down or bottom-up fashion, starting with either the symbolic or numerical component first. At some point along either route, the interface between STAR and the general-purpose language will come into consideration. The driving notion behind the interface apparatus existing in STAR is that an effective communication between routines and data structures in one language and routines and data structures in another language relies on two components:

(1) Both languages should be able to access pointers to data structures defined in the other language. These pointers may then be stored by each language in combination with its own data structures.

(2) Each language should provide a set of utility routines callable by the other language which allow the other language to perform basic operations on the data structures of the first language. This implies that first of all, it must be possible for both languages to call functions in the other language.

There are two directions of exchange in the interface, that of STAR accessing data structures from the general-purpose language and performing operations upon them and

that of the general-purpose language accessing STAR UNITs and performing operations upon these structures. An overview of both of these directions of communication is given in the remainder of this section.

The CONNECTION type of UNIT has been introduced in Part I of this guide. CONNECTIONs are pointers to either functions or data structures defined in compiled programming languages. A lookup table modified by the application programmer allows STAR to recognize an initial set of functions defined in the general-purpose languages. Definitions for these functions are automatically generated in STAR, and they appear as members of the class "function" or various subclasses of "function" in the STAR knowledge base. Initially, no external data structures are known to STAR, but it is possible for external functions to return CONNECTIONs to external data structures as results and thus they may enter the STAR environment in this manner. A simple example illustrating some of these mechanisms appears below. The details of how to provide STAR with an initialized set of external functions are covered in a later section.

**Example 29:**

```
    $ Assume previously defined: "c_function",
    $   "image", "data", "fetch_image_1" and
    $   "image_pixel".

    fetch_image_1                   $ Suppose "fetch_image_1" is
                                    $   initialized within STAR.
{name -> FETCH_IMAGE_1
 member_of -> c_function
 comment ->
     "     () => CONNECTION
     "
     "     Returns a CONNECTION to a two-dimensional
     "array defined in C."
 n_arguments -> 0
 algorithm -> ^FETCH_IMAGE_1_FUNCTION
 }

    image_pixel                     $ Suppose "image_pixel" is also
                                    $   initialized.
```

```
{name -> IMAGE_PIXEL
 member_of -> c_function
 comment ->
     "     (CONNECTION1 NUMBER1 NUMBER2) => NUMBER
     "
     "     Given CONNECTION1, pointing to a two-
     "dimensional array defined in C, and the pair
     "NUMBER1 and NUMBER2, specifying line and
     "sample coordinates, return the designated
     "pixel value."
 n_arguments -> 2
 algorithm -> ^IMAGE_PIXEL_FUNCTION
 }

    create(image_1 image)     $ A new named RECORD to store
                              $   the image within STAR.
{name -> IMAGE_1
 member_of -> image
 }

    modify(                   $ Accessing a pointer to the
     image_1                  $   image through calling the
     data                     $   function "fetch_image_1".
     fetch_image_1()          $   The new CONNECTION is
     )                        $   stored with the attribute
                              $   "data" within "image_1".
{name -> IMAGE_1
 member_of -> image
 data -> ^ORIGINAL_IMAGE
 }

    image_pixel(              $ Using the stored reference
     get(image_1 data)        $   to the C image array,
     100 34                   $   extract the pixel value
     )                        $   at (100,34).

212

    image_pixel(              $ Extract the pixel value at
     get(image_1 data)        $   (110,34).
     110 34
     )

230
```

The above example illustrates both how pointers to external data structures may be positioned within the the STAR knowledge base and how these CONNECTIONs may then be used as arguments when calling the external functions which operate upon their contained data structures. In the above, CONNECTIONs are inserted into the STAR knowledge base following their creation in external functions and returned to STAR as results of the external function calls. Another method for inserting CONNECTIONs into the STAR knowledge base involves the sending of UNITs such as LISTs or named RECORDs as arguments to special external functions which create CONNECTIONs and incorporate these into the UNITs directly. This method relies on capabilities described later in this section.

At this point, it may be informative to glance at the other side of the interface. The functions "fetch_image_1" and "image_pixel" in the above example take STAR UNITs as arguments and return STAR UNITs as results. Within general-purpose languages, STAR UNITs are treated as integer values and operated upon primarily by a set of approximately 50 utility functions defined in C and accompanying the STAR interpreter code. Using these utility functions, it is possible to extract the actual numerical values or character strings from STAR NUMBERs, TOKENs and STRINGs, to extract elements or incrementally step through the structures of STAR LISTs, RECORDs and EXPRESSIONs, and to create or modify UNITs of all types in basic ways. The following example specifies how the functions corresponding to "fetch_image_1" and "image_pixel" might be defined in C.

**Example 30:**

```
typedef int unit;

unit fetch_image_1()
  {
  extern char *image_1;
  extern unit make_connection();
  return(make_connection(image_1,"ORIGINAL_IMAGE"));
  }

unit image_pixel(con1,num1,num2)
unit con1,num1,num2;
  {
  extern int get_connection_contents();
  extern double get_number();
  extern unit make_number();
  char *image;
  int line,sample;
  image = (char *) get_connection_contents(con1);
  line = (int) get_number(num1);
  sample = (int) get_number(num2);
  return(
   make_number(
```

```
    (float) image[(line-1)*512 + (sample-1)]
    ));
}
```

Note in the above example the use of a dummy type "unit" which is equivalent to type "int" (integer). This is the simplest way to keep track of which data values are in fact pointers to STAR UNITs, while at the same time isolating the external functions from the internal workings of UNIT structures. The same declaration strategy may be used for external functions defined in other general-purpose languages where this is allowed by the language in question.

The functions "make_connection", "get_connection_contents", "get_number" and "make_number" in the above example are all STAR utility functions. These functions are summarized here, although the entire set of STAR utility functions is covered in much greater detail in the next section and in Section 6 of the Reference Manual. In the operation of "fetch_image_1", the function "make_connection" takes as arguments the data value to be stored within a new CONNECTION (this is the value of "image_1" in the above example) plus a character string specifying the label for the CONNECTION. In this manner, "fetch_image_1" simply accesses the image address, forms a CONNECTION specifying this value and returns the CONNECTION to STAR. The operation of "image_pixel" is somewhat more involved. Since the arguments sent to "image_pixel" are pointers to STAR UNIT structures, the data values of interest within "image_pixel" must be extracted, here by the STAR utility functions "get_connection_contents" and "get_number". Once the intended data values have been extracted, the image subscripting operation is straightforward, with a call to "make_number" used to form a STAR NUMBER for return to STAR.

The reader may have wondered how it is that STAR is able to communicate both with languages which pass arguments "by value", such as C, and languages which pass arguments "by reference", such as FORTRAN and sometimes PASCAL. This distinction has been omitted in the above examples. For each of the STAR utility functions, STAR actually contains two versions, one for each type of interface. In the above example, the "by value" versions of the required utility functions are used. For external functions defined in FORTRAN, for example, the "by reference" versions of utility functions would be used. (Due to constraints imposed by some FORTRAN compilers, names for the "by reference" versions are shortened to three letters per word with no contained underscore characters: e.g., "makcon" is the "by reference" version of "make_connection", "getconcon" the "by reference" version of "get_connection_contents", and so forth.) In addition, in the initialization table for declaring external functions, each function is specified as to whether arguments are to be sent by value or by reference. This takes care of both directions of communication: arguments sent by STAR to an external function are sent by whatever method is designated for that function in the initialization table, and the choice of STAR utility functions used by an external function determines intrinsically the method of passing arguments from the general-purpose language to STAR in this case.

As described thus far, the interface between STAR and a general-purpose language lacks one important feature: the ability for functions defined in general-purpose languages to call not only the STAR utility functions, but the actual built-in and user-defined

functions of STAR itself. This is possible as well. Since the built-in functions of STAR are implemented each as a function defined in C, one need only call the appropriate C function for a STAR built-in function. Two versions exist for each of the defining C functions (only the "by value" versions are actually used within the interpreter operation itself). To call a STAR built-in function from a language which passes arguments by value, the C function of the same name as the built-in function but ending in "_f" is used (e.g., "add_f" is called for the "add" function, etc.). For languages which pass arguments by reference, the function ends in simply "f" (e.g., "addf" for the "add" function). No underscore is used in the "by reference" version as some FORTRAN compilers do not permit the use of this character in symbolic names. From the point of view of external functions, all STAR built-in functions must be sent arguments which correspond to pointers to UNIT structures, and the result values returned by these functions are of the same type.

To call user-defined STAR functions from external functions, the "apply" built-in function of STAR is used. This is because user-defined functions are not compiled in STAR and thus do not exist in any form which can be found in the symbol table of the STAR interpreter code. The following example illustrates the calling of STAR built-in and user-defined functions from within external functions. This example is written in FORTRAN and involves a function which takes pointers to two STAR LISTs as arguments, calls the built-in STAR function "union" to form a LIST representing the set union of the elements of these LISTs, and then calls a user-defined STAR function "sort" to sort the LIST according to some ordering function.

**Example 31:**

```
integer function unionsort(lis1,lis2)
integer lis1,lis2,lis3,lis4,unionf,applyf
integer getnamrec,maklis,inslisattai
external unionf,applyf
external getnamrec,maklis,inslisattai
lis3 = unionf(lis1,lis2)
lis4 = inslisattai(maklis(),lis3)
unionsort = applyf(getnamrec("sort"),lis4)
return
end
```

The call to "apply" in the above example involves the use of three STAR utility functions. Since FORTRAN passes arguments by reference, the "by reference" versions of these three utility functions have been used. The function "getnamrec" ("by reference" version of "get_named_record") takes a character string as its argument and returns the STAR named RECORD of that name. The function "maklis" ("by reference" version of "make_list") creates a zero element LIST and returns it. This LIST is then used as an argument to "inslisattai" ("by reference" version of "insert_list_at_tail"), which places the union LIST ("lis3") within the newly created LIST. These functions must be used as "apply" requires as its second argument a STAR LIST of all arguments to be sent to the function in question.

A number of methods have been introduced in this section both for calling functions between languages and passing data structures between languages. It is therefore useful to summarize the available techniques below. An external function may be called from within STAR by the following.

(1) Preparing an entry for the function in the initialization table of the STAR interpreter. The function then appears as a STAR function, being a member of "function" or a subclass of "function". The function may then be called directly as any other function in STAR.

External data structures may be passed into STAR by the following.

(1) Creating special external functions which construct STAR CONNECTIONs to these data structures and return the CONNECTIONs as results. An example is the function "fetch_image_1" depicted in Example 30.

(2) Sending STAR UNITs such as LISTs or named RECORDs as arguments to external functions, and allowing the external functions to insert the CONNECTIONs directly, aided by various STAR built-in functions and utilities as called from within the external functions.

STAR functions may be called from within external functions by the following.

(1) For a STAR utility function, choosing between the "by value" and "by reference" versions of the function and calling the function directly from the external function. Care must be taken to properly declare the results returned by utility functions, as they may vary from pointers referencing STAR UNITs to integers, double precision floating point numbers or character strings.

(2) For a built-in STAR function, choosing between the "by value" and "by reference" versions of the function and calling directly from the external function, in the same manner as for calling utility functions. The "by value" versions of built-in functions are suffixed with "_f", while the "by reference" versions are suffixed with "f".

(3) For user-defined STAR functions, it is necessary to use the built-in function "apply" to perform the call indirectly. An example of this appears in the definition of "unionsort" in Example 31.

STAR UNITs may be conveyed to the external functions by the following.

(1) As arguments when calling the external functions from STAR.

(2) As return values from STAR functions of all types when called from the external functions.

In the above discussion, one additional interface capability has been passed over. This involves the use of the built-in function "system" as a means of calling separate executable files or otherwise setting up concurrent processes within the host computer. In this case, a limited capability for passing arguments to the other process exists in the use of command-line arguments when calling the process. Otherwise, it is perhaps most practical to pass data values through files accessible to both processes. This means of interface has been underplayed for two reasons: (1) the capability is not entirely portable or general, as individual operating systems differ and may not support this capability, and (2) the intercommunication of data values between processes by means of files requires somewhat more overhead than the combination of executable code into a single running process and the intercommunication of data values by means of CONNECTIONs and other UNIT structures.

## 17. The STAR Utility Functions

As noted in the previous section, a set of approximately fifty compiled utility functions accompanies the STAR interpreter code. These functions are made available to external functions linked with the STAR interpreter and are used in the creation, access and modification of STAR UNIT structures. Also, external functions may call the built-in functions of the STAR language and, indirectly through the use of the "apply" function, user-defined functions defined in STAR. This section is concerned exclusively with the use of the STAR utility functions.

To facilitate the manipulation of UNIT structures by external functions defined in languages which pass arguments by value and in languages which pass arguments by reference, two versions exist for each of the utility functions. A simple rule specifies the distinction between the "by value" version of a utility function and its corresponding "by reference" version: utility functions which take arguments by value possess names in which the individual words are separated by the underscore character ("_"), whereas utility functions which take arguments by reference possess names in which the individual words follow each other directly and additionally utilize only the first three letters of each word. (This naming scheme complies with some FORTRAN compilers which do not allow underscore characters in symbolic names or which limit symbolic names to under 16 characters.) For example, the functions "get_number", "make_list" and "insert_current_record_entry" take arguments by value, while their counterparts "getnum", "maklis" and "inscurrecent" take arguments by reference. The distinction is apparent in the examples which follow: all FORTRAN examples use the "by reference" versions of utility functions, while all C examples use "by value" versions. The language PASCAL may pass arguments by value or by reference, depending on the implementation. For convenience, a "by value" convention for PASCAL is assumed within this manual,

although either variety may be interfaced with STAR.

### Simple Access and Modification Utilities

The STAR utilities may be divided into two subsets according to complexity. The simpler set involves functions which perform basic conversions between UNIT structures and conventional data structures, create UNIT structures and modify these structures in basic ways. The more complicated set involves functions which perform incremental operations on LISTs, RECORDs and EXPRESSIONs, allowing these structures to be examined and modified one element at a time.

It is important to note from the start that the STAR utility functions operate at a fairly low level compared to the built-in functions of STAR and thus the programmer must take on an added degree of responsibility in assuring that no ill side-effects are produced. As an example of this, the UNITs sent as arguments to external functions should be checked as to their type immediately upon receipt. The utility function "get_unit_type" ("by reference" form "getunityp") provides this capability, taking a single UNIT as its argument and returning a coded integer corresponding to the UNIT's type. The coded values are as follows.

1 -- NUMBER,
2 -- TOKEN,
3 -- STRING,
4 -- LIST,
5 -- RECORD,
6 -- EXPRESSION,
7 -- CONNECTION.

The utility "get_unit_usage_count" indicates whether or not a particular UNIT is presently being shared, or multiply-referenced as elements in other UNITs. If it is desired to modify a UNIT which is presently being shared and the modifications are not to be carried over to other references to that UNIT, the UNIT should be copied first using the utility "copy_unit" (copies an entire NUMBER, TOKEN, STRING or CONNECTION or the top level of a LIST, RECORD or EXPRESSION). Another utility, "get_unit_protection_code", indicates whether a UNIT belongs to the initialized portion of the STAR knowledge base (i.e., modification of the UNIT within STAR would result in a PERMISSION ERROR). While it is permitted to perform operations on initialized UNITs using the STAR utilities, extreme care must be taken in doing so and the programmer should possess full knowledge of any consequences produced by the modifications. Other responsibilities which must be assumed at this level involve named RECORDs and are described later in this section.

Two utilities are provided for each of the simple UNIT types, NUMBERs, TOKENs and STRINGs. These functions perform conversions from the UNITs to conventional data structures and vice versa. For NUMBERs, the utility "get_number" takes a single NUMBER as its argument and returns a double precision floating point value (the value is double precision due to constraints imposed by the C language). The function

"make_number" takes a single-precision floating point value and returns a newly-created STAR NUMBER corresponding to that value. The corresponding utilities for TOKENs and STRINGs are analogous: "get_token" and "get_string" convert TOKENs and STRINGs to conventional character strings, and "make_token" and "make_string" form TOKENs and STRINGs given conventional character strings as arguments. The following example illustrates the implementation of a simple function in PASCAL ("by value" version assumed) which uses the utilities "get_unit_type", "get_string" and "make_number" in counting the number of equal corresponding characters, position by position, within two 32-character STRINGs.

**Example 32:**

```
type unit = integer;
type chstring = array [1..32] of char;

function get_unit_type(u:unit) : integer;
  external;

function get_string(s:unit) : chstring;
  external;

function make_number(f:real) : unit;
  external;

function count(str1,str2:unit) : unit;
var s1,s2: chstring;
    i: integer;
    c: real;
begin
  count := make_number(0.0);
  if (get_unit_type(str1) = 3)
    and (get_unit_type(str2) = 3) then
  begin
    s1 := get_string(str1);
    s2 := get_string(str2);
    c := 0.0;
    for i := 1 to 32 do
      if s1[i] = s2[i] then c := c+1;
    count := make_number(c)
  end
end;
```

Note that the external declaration for "get_string" above requires an explicit size constraint in PASCAL (fixed 32-element character strings). A similar case exists in FORTRAN. Unfortunately, if STRINGs or TOKENs of varying sizes are to be conveyed to languages with strict data typing constraints on the lengths of character strings, it is

necessary to determine an upper limit on the length of all character strings to be used and declare "get_string" or "get_token" within these languages as returning character strings of length corresponding to that upper limit. Care must be taken in such cases that the actual end of the character string is not exceeded in the compiled languages. An alternative approach is built into STAR for the utility "get_connection_contents": STAR contains in all a set of ten identical versions of this utility, allowing each version to be declared as returning a value of a different type. The programmer may find it useful to define (in C) redundant copies of "get_string" or "get_token" for similar purposes if necessary.

There are five STAR utilities of the simpler variety associated with LISTs. These are "get_list_size", "get_list_element", "make_list", "insert_list_at_head" and "insert_list_at_tail". The use of these functions is illustrated in the following example, involving a C function which takes a two-element LIST, forms a new two-element LIST with the elements reversed, and returns this second LIST to STAR.

**Example 33:**

```
typedef int unit;

unit reverse(lis1)
unit lis1;
 {
 extern int get_list_size();
 extern unit get_list_element();
 extern unit make_list();
 extern unit insert_list_at_head();
 extern unit insert_list_at_tail();
 unit result;
 result = make_list();
 if(get_unit_type(lis1) != 4) return(result);
 if(get_list_size(lis1) != 2) return(result);
 result = insert_list_at_head(
            result,
            get_list_element(lis1,2)
            );
 result = insert_list_at_tail(
            result,
            get_list_element(lis1,1)
            );
 return(result);
 }
```

The utilities "insert_list_at_head" and "insert_list_at_tail" insert the specified element at the indicated end of the LIST appearing as the first argument, and return the modified LIST. Note that the LIST is not copied first, but is modified directly by this process.

The simpler utility functions for RECORDs and EXPRESSIONs are largely analogous to those for LISTs. Following is a list of these utilities; descriptions may be found in the STAR Reference Manual.

| | |
|---|---|
| get_record_size, | get_expression_size, |
| get_record_attribute, | get_expression_operation, |
| get_record_value, | get_expression_argument, |
| match_record_value, | |
| make_record, | make_expression, |
| get_named_record, | |
| insert_record_at_head, | insert_expression_at_head, |
| insert_record_at_tail, | insert_expression_at_tail. |

The remaining five of the simpler STAR utilities involve the handling of CONNECTIONs. The utilities "get_connection_contents" and "get_connection_label" access, respectively, the data value stored in a CONNECTION and the character string labeling the CONNECTION. The utility "make_connection" takes a data value and a character string and forms a new CONNECTION. The utilities "reassign_connection" and "relabel_connection" allow direct modifications to be made to the contents and labels of existing CONNECTIONs. The following example illustrates a PASCAL function which uses two of these utilities. The function extracts from the CONNECTION sent as its argument a PASCAL record describing a color in terms of red, green and blue components. The function "darkens" this color by forming a second PASCAL record with red, green and blue values scaled to 70 percent of their original values. This second PASCAL record is housed in a new CONNECTION which is returned to STAR.

**Example 34:**

```
type unit = integer;
type chstring = array [1..32] of char;

type color = record
  red   : 0..255;
  green : 0..255;
  blue  : 0..255
end;
type pcolor = ^color;

function get_named_record(s:chstring) : unit;
  external;

function get_unit_type(u:unit) : integer;
  external;
```

```
function get_connection_contents(c:unit) : pcolor;
  external;

function make_connection(c:pcolor; s:chstring) : unit;
  external;

function darken(con1: unit) : unit;
var oldcolor,newcolor : pcolor;
begin
  darken := get_named_record('nil');
  if(get_unit_type(con1) = 7) then
  begin
    oldcolor := get_connection_contents(con1);
    new(newcolor);
    newcolor^.red := (7 * oldcolor^.red) div 10;
    newcolor^.green := (7 * oldcolor^.green) div 10;
    newcolor^.blue := (7 * oldcolor^.blue) div 10;
    darken := make_connection(
                  newcolor,
                  'COLOR_@'
                  )
  end
end;
```

Note the use of the utility "get_named_record" in the above example. This is a convenient way to access standard response values such as "nil", "true" and "false".

In the character string pattern sent as the second argument to "make_connection", the following conventions are used in forming a label for the new CONNECTION. Any contained "at" characters ("@") in the pattern are converted to an ASCII representation of the address for the data structure or routine to be placed in the "contents" field of the CONNECTION. Any lowercase letters in the pattern are converted to uppercase. Uppercase letters and all digits are left as they are and all other characters are converted to the underscore ("_") character. Thus, the final label for the CONNECTION contains only capital letters, digits and instances of the underscore character. For instance, the CONNECTION formed in the above example might appear as "^COLOR_1173408", where 1173408 is the address of the pointer to "color" used as the data contents for the CONNECTION.

The reader may notice certain restrictions placed on the declarations for "get_connection_contents" and "make_connection" in the above example. Specifically, these declarations have required precise tailoring to the "pointer to color" type stored in the CONNECTIONs. In C, for example, such declarations are not restrictive as it is possible to "cast" data values of one type into data values of another type. For languages with strict data typing such as PASCAL and FORTRAN, STAR includes a set of ten redundant copies for each of the utilities "get_connection_contents", "make_connection" and "reassign_connection" (see Section 6 of the Reference Manual). The copies for get_connection_contents", for example, are named "get_connection_contents" and

"get_connection_contents_1" through "get_connection_contents_9" for the "by value" versions and "getconcon" and "getconcon1" through "getconcon9" for the "by reference" versions. Within the compiled language, each version of a utility may be declared as taking arguments of different types or returning a result value of a different type.

### Incremental Access and Modification Utilities

For some types of operations, it is desirable to be able to step incrementally through a LIST, RECORD or EXPRESSION, maintaining a stored reference to points internal to these structures so that the operation may continue without requiring a long chaining from the beginning of the structure at each new step. Compare this to the operation of a utility such as "get_list_element", described above. When this utility is called with a LIST and an integer element number, it locates the desired element by starting at the beginning of the data structure implementing the LIST and stepping through the specified number of elements. If "get_list_element" is called several times with a LIST and different element numbers, it steps through the LIST separately for each call.

A set of seven STAR utilities for LISTs, eight for RECORDs and seven for EXPRESSIONs provides a mechanism for performing incremental operations. For LISTs, the utilities are "begin_list_scan", "end_list_scan", "get_current_list_element", "get_next_list_element", "insert_current_list_element", "insert_next_list_element" and "remove_current_list_element". The operation of the functions is basically as follows. Several concurrent incremental operations are possible on the same LIST; each is initialized by calling "begin_list_scan" with the desired LIST and a distinguishing integer (e.g., "3", "17", etc.) as arguments. The integer labels a particular incremental sweep or "scan". Once "begin_list_scan" has been called, the utility "get_next_list_element", when given the LIST and the labeling integer as arguments, may be called to retrieve the elements of that LIST, one at a time. That is, each time "get_next_list_element" is called, it returns the next element. The utility "get_current_list_element" may also be called to reexamine a given element without progressing in the scan.

The utilities "insert_current_list_element", "insert_next_list_element" and "remove_current_list_element" allow additions and deletions to be performed within a LIST during a scanning process. When "insert_current_list_element" is called, a new UNIT is inserted into the LIST directly before the element which would be returned by "get_current_list_element". The new UNIT also becomes the new current element. When "insert_next_list_element" is called, a new UNIT is inserted directly following the current element, with the current element remaining as it was. When "remove_current_list_element" is called, the current element is deleted from the LIST, and a state is entered wherein there is no longer a "current" element, yet it is still possible to proceed to the next element or insert a current or next element. The following example illustrates the use of some of these utilities. It involves a FORTRAN function which forms a STAR LIST specifying the powers of two from 1 to 1024.

### Example 35:

```
C "By reference" utilities translate as follows:
C       beglissca   = begin_list_scan
C       endlissca   = end_list_scan
C       inslisathea = insert_list_at_head
C       insnexlisele = insert_next_list_element
C       maknum      = make_number
C       maklis      = make_list
C       getnexlisele = get_next_list_element
C       getnum      = get_number

      integer function powers()

      integer beglissca,endlissca,inslisathea
      integer insnexlisele,maknum,maklis,getnexlisele
      double precision getnum
      external beglissca,endlissca,inslisathea,getnum
      external insnexlisele,maknum,maklis,getnexlisele

      lis = maklis()
      lis = inslisathea(lis,maknum(1.0))
      lis = beglissca(lis,1)
      do 10 i = 1,10
      x = getnum(getnexlisele(lis,1))
10    lis = insnexlisele(lis,1,maknum(x * 2))
      lis = endlissca(lis,1)
      powers = lis
      return
      end
```

The operation of the function "powers" in the above example proceeds as follows. First, an empty list is created by "maklis", into which a single NUMBER "1.0" is inserted. This being the value of "lis" at the start of the DO loop, each cycle extracts a single element of the LIST, doubles its numerical value and inserts this as a new LIST element following the current one. The process thus keeps one step ahead of itself in inserting new elements before they are retrieved.

A second example of the incremental STAR utility functions is given below, this time involving a PASCAL function which receives a LIST of NUMBERs as its argument and returns the LIST with all occurrences of the NUMBER "3" removed.

**Example 36:**

```
type unit = integer;

function begin_list_scan(l:unit; i:integer):unit;
  external;

function get_next_list_element
        (l:unit; i:integer) : unit;
  external;

function get_current_list_element
        (l:unit; i:integer) : unit;
  external;

function get_number(l:unit):real;
  external;

function remove_current_list_element
        (l:unit; i:integer) : unit;
  external;

function end_list_scan(l:unit; i:integer):unit;
  external;

function remove_3s (l: unit) : unit;
begin
  l := begin_list_scan(l,1);
  while(get_next_list_element(l,1) <> 0) do
    if get_number(get_current_list_element(l,1)) = 3
      then l := remove_current_list_element(l,1);
  l := end_list_scan(l,1);
  remove_3s := l
end;
```

It may be useful to trace the operation of "remove_3s" in the above example. Suppose this function is called with a STAR LIST such as "[3 2 5 3 3 1]" for its argument. The call to "begin_list_scan" initiates scan "1" for the LIST "l". At this point, there is no current element as of yet, and thus the first call to "get_current_list_element" retrieves the first element of the LIST. Since the function "get_next_list_element" returns a value of type "unit" which is equated to "integer" in PASCAL, a null value corresponds to the integer "0". If the LIST contained no elements, the body of the while loop would never be executed.

Inside the while loop, the current element is tested to see if its numerical value corresponds to the NUMBER "3". If so, the function "remove_current_list_element" is called, removing the NUMBER "3" from the LIST "l". As noted above, calling the



C-2

function "remove_current_list_element" results in a state in which there is no longer a current element, yet it is still possible to proceed to the next element, insert a new current element or insert a next element. A useful visualization of this state is to imagine the current pointer to the LIST positioned in the "space" between the two elements (now adjacent) which previously surrounded the removed element. If "get_next_list_element" is called, the element following the one removed becomes the current element. If "insert_current_list_element" is called, a new UNIT is inserted where the old element was removed and this becomes the new current element. If "insert_next_list_element" is called, a new UNIT is inserted following the "space" pointed to, but this does not become the current element. The internal pointer may be visualized at this point as indicating the "space" between its previous preceeding element and the new UNIT just inserted. In the case of "remove_3s", above, the call to "remove_current_list_element" is followed by a call to "get_next_list_element", and thus the scanning operation continues with the next element in the LIST. In this manner, the scanning operation is not interrupted in its progress by the removing of an element. In the above instance of the LIST "[3 2 5 3 3 1]" being sent as an argument to "remove_3s", the result would be the LIST "[2 5 1]", following six iterations of the while loop and three calls to "remove_current_list_element".

The incremental access and modification utilities in STAR provide a powerful and efficient framework for operating on the structures of the STAR knowledge base outside of STAR. Since each scanning operation is made distinct by the inclusion of a labeling integer, it is also possible to conduct multiple simultaneous scans on a particular structure. The utilities are designed so that their use in combination in this manner will not conflict one with another. For instance, two scans on a LIST may be initiated and both may add or delete elements from the same LIST. Of course, the changes effected on a first scan which precedes a second scan will be reflected in the elements visited by that second scan. The only case for direct conflict or interference between scans which arises is when two or more scans point to the same element within a structure, and one scan removes the indicated element. In this case, all of the scans which indicated the element in question will progress to the "in between" state of indicating no current element.

As with the simple access and modification utilities, the incremental access and modification utilities for LISTs are reflected in closely analogous sets of utilities for RECORDs and EXPRESSIONs. Following is a list of these utilities. Specific explanations for their use may be found in the Reference Manual.

| | |
|---|---|
| begin_record_scan, | begin_expression_scan, |
| end_record_scan, | end_expression_scan, |
| get_current_record_attribute, | get_current_expression_argument, |
| get_current_record_value, | |
| get_next_record_attribute, | get_next_expression_argument, |
| insert_current_record_entry, | insert_current_expression_argum, |
| insert_next_record_entry, | insert_next_expression_argument, |
| remove_current_record_entry, | remove_current_expression_argum. |

(Names for the utilities "insert_current_expression_argum" and

"remove_current_expression_argum" have been truncated to 31 characters due to restrictions imposed by certain C compilers.)

The incremental access and modification utilities for RECORDs and EXPRESSIONs are used in the same manner as those for LISTs with one exception. Since the modification of named RECORDs is in some cases intended to produce side effects (as with modification of entries for the attributes "name", "member_of" and "subclass_of"), it is recommended to use built-in STAR functions such as "assert", "modify" and "retract" in cases where side-effects are desired rather than the incremental access and modification utilities, which do not produce side effects for changes made in the course of their operation.

## 18. Combining the Various Parts

At this point, the various parts of an application system have been described in some detail, but the integration of these parts into a complete operational system has not yet been fully covered. The parts to be integrated include functions defined and compiled in general-purpose languages, STAR definitions appearing in text format on various files and libraries of associated routines, say a mathematics package or graphics display package. To see how the integration is accomplished, it is necessary to consider first the layout for the STAR interpreter source code.

The STAR interpreter is defined in the language C and is separated into a set of 14 source files. The names of these files, along with brief summaries of their use, are given below. Most of these files are simply compiled and linked together with the rest of the code for an application system. However, the file "starlink.c" has a particular usage in the context of application systems and the application programmer will need to become somewhat familiar with its layout and contents.

**stardefs.h, starcomm.h:** These files contain type definitions and constants shared between the various source code files. The C command "include" is used to initialize the STAR source files with these definitions.

**starcode.c:** Contains the major portion of the source code for the STAR interpreter mechanism.

**starbifs.c:** Definitions for the built-in functions of STAR. These are implemented each as a function in the language C.

**starini1.c, starini2.c, starini3.c, starini4.c, starini5.c, starini6.c:** These files contain initialized data structures specifying the original state of the STAR knowledge base, including all named RECORDs contained in the knowledge base at startup time for STAR.

**starplus.c:** Definitions for built-in functions "system", "suspend" and "exit" which are more machine- or application-dependent in nature. Also, contains "by reference" versions of all STAR built-in functions, to be called by external functions defined in languages which pass arguments by reference.

**starhack.c:** Machine-dependent initialization code separated out of the file "starcode.c".

**starutil.c:** Definitions for the STAR utility functions, along with descriptions of their use and argument/result type specifications.

**starlink.c:** Initialization tables for classes of external functions and individual members of those classes.

As the external functions and data structures defined for a particular application must coexist with the functions and data structures of the STAR interpreter, all symbolic names in STAR (with the exception of the STAR utilities, which possess sufficiently unique names) are coded so as to minimize the occurrence of name conflicts with external structures. Within the interpreter code, all type names end in "_t", global variables in "_g", built-in function definitions in "_f" and "f", and supporting function definitions in "_s". In most cases, then, the names of external functions and data structures should not find interference elsewhere within STAR.

The general approach for integrating the parts of a particular application system in STAR is as follows. First, the application programmer modifies the file "starlink.c" to specify names for classes of external functions and the individual members of those classes. Next, the compiled version of "starlink.c" is linked together with the compiled modules for the remainder of the STAR interpreter, compiled modules for any external functions indicated in "starlink.c", and library packages as needed. The result is an executable file named "star", which is an enhanced version of the STAR interpreter allowing the programmer to call the specified external functions exactly as he would call the built-in functions of STAR.

The next step involves the symbolic definitions associated with an application system. Typically, these are user-defined functions in STAR, modifications to the initial STAR knowledge base and definitions for rules and other control structures. Such definitions may be stored in text files and included within the STAR environment through use of the "load" built-in function. As such definitions are often quite extensive in nature, however, it would be cumbersome to require these to be loaded in each time the application system was to be used. The "suspend" built-in function in STAR obviates this repetitive task. Following the loading of several text files specifying STAR definitions, a call to "suspend" will form a new executable file corresponding to the STAR interpreter linked with associated external functions and also enhanced with the loaded symbolic definitions. This new executable file may then be used in later sessions, resulting in an immediate entry from the operating system into an environment which includes the STAR interpreter, the specified external functions and all symbolic definitions loaded into the interpreter prior to the call to "suspend". The application system may be entered in a two-step process, by

first entering STAR, then calling a predesignated user-defined function or rule class which invokes the application mechanism, or the application system may be entered in a single step, using a set of special arguments in the operating system call to STAR. This latter course is described in detail at the end of this section.

With the general process of forming an application system described in the above paragraphs, the mechanics of carrying out the indicated steps are covered below. One aspect to be considered is whether one or several programmers are to be developing applications using STAR. In the case of one programmer, the easiest arrangement is perhaps to maintain a file directory containing object code files for all of the STAR source code except "starlink.c", which is maintained in source form. Also in this directory would be source and object files for external functions and text files for symbolic definitions to be loaded into STAR. Whenever a change is made in the external functions to be linked to STAR, the change is recorded in the proper format in "starlink.c", "starlink.c" is compiled, and all object files for STAR and the external functions are relinked, followed by the reloading of symbolic definitions and formation of a new top-level executable file using "suspend".

In the case where more than one programmer is developing application systems using STAR, it may be simpler to compile all source code for the STAR interpreter with the exception of "starlink.c", link the object files together and place the resultant object file in a place generally accessible to the programmers involved. Each programmer then maintains his own local copy of "starlink.c" and any external function definitions and/or symbolic definitions involved in his application. The major difference is that the linking process is performed using a central copy of the main STAR interpreter code. In this case, each programmer still ends up with his own copy of the STAR interpreter in executable form once the main body of STAR code has been linked together with his particular application-dependent code.

The modifications to be made within "starlink.c" are intended to be of minimal complexity. Since this file is to be compiled in C, however, some restrictions regarding the C syntax apply. The file "starlink.c" is small in size and thus compiles quickly once modified. Contained in this file are (1) a table describing classes of external functions, (2) a set of external function declarations for the C compiler, and (3) a table describing the individual external functions in detail.

The table specifying classes of external functions is considered first. Individual external functions may either be specified as belonging to the class "function" or to specific subclasses of "function" to be created at initialization time. If there are no external functions for a particular application or if the external functions are all intended as direct members of the class "function", the table of external function classes will be empty. Otherwise, if it is desired for external functions to be placed as members of one or more new subclasses of "function", specific entries are made in the table describing the new subclasses.

The following example illustrates entries made in the global table for external function classes for two classes of functions, "fortran_function" and "pascal_function", the members of which are considered later. The table for classes of external functions is called "external_function_classes_g" and appears in the part labeled "SECTION I: ..." in the file. To distinguish the lines inserted by the application programmer from the "shell" of the

table existent in "starlink.c" prior to the modifications, a set of asterisks ("*") have been placed preceding the lines entered by the programmer. The asterisks are included only for illustration and do not appear in the actual modifications to "starlink.c". Vertical bars indicate the left margin of the file.

**Example 37:**

```
|
|/* SECTION I: CLASSES OF EXTERNAL FUNCTIONS. ...*/
|...
|
|struct external_function_class_entry_t
|  external_function_classes_g[] =
|...
|  {
|
*|    {"fortran_function",
*|
*|  "     Functions defined in FORTRAN and linked\
*| \nto the STAR interpreter."},
|
*|    {"pascal_function",
*|
*|  "     Functions defined in PASCAL and linked\
*| \nto the STAR interpreter."},
|
|    {0,0} /* "End of table" entry.  Do not remove. */
|  };
|
```

Each entry contains two elements: a class name and a comment string. The class name appears in lowercase and surrounded by double quotes. The name provided specifies the function's reference name in STAR; thus, it must start with a lowercase letter and may contain lowercase letters, digits and underscore characters. The comment string is an arbitrary sequence of characters surrounded by double quotes, with restrictions according to the syntax of C character strings. In order to specify a carriage return within a C character string, the sequence "\<cr>\n" must be used, where "<cr>" is an actual carriage return. This has the effect of forcing C to ignore the actual carriage return while accepting the code "\n" which specifies a carriage return for inclusion in the string. Compiling errors result if the carriage returns in comment strings are not entered in this manner. The two elements of the entry are separated by a comma and surrounded by left and right brace characters ("{" and "}"), and the entry is followed by a comma (omission of the comma also results in compiling errors).

The entry "{0,0}" appearing at the end of the table definition is a dummy entry signifying the end of the table. This entry must remain in the definition of "external_function_classes_g" regardless of the modifications performed.

Next, suppose that as members of the above classes it is desired to use the functions "darken", "powers" and "remove_3s", as defined in Examples 34, 35 and 36 in Section 17, along with an additional C function "specification". The function "powers" is to belong to the class "fortran_function", "darken" and "remove_3s" to "pascal_function" and "specification" directly to "function".

The next step in the modification of "starlink.c" is the entry of a set of external function declarations for the C compiler. This is done in the section marked "SECTION II: ..." in the file. The following example illustrates the set of declarations for the four external functions "powers", "darken", "remove_3s" and "specification". Again, asterisks are used to mark the lines entered by the application programmer, and vertical bars are used to indicate the left margin of the file.

**Example 38:**

```
  |
  |/* SECTION II: EXTERNAL FUNCTIONS -- DECLARATIONS. ...*/
  |...
  |
* |extern struct unit_t *powers_();
* |extern struct unit_t *darken();
* |extern struct unit_t *remove_3s();
* |extern struct unit_t *specification();
  |
```

Each entry follows the same format: "extern struct unit_t *xxx();", where "xxx" is the name of the function. One difference exists, however, for FORTRAN functions in some cases, depending on the particular FORTRAN compiler in use. If the name of the function as defined in FORTRAN does not work in the context of "starlink.c", it may be necessary to affix a single underscore character to the end of the name, as has been done for "powers", above.

The final step in the modification of "starlink.c" is the insertion of individual entries in the table of external functions. This table is named "external_functions_g" and appears in the part labeled "SECTION III: ..." in the file. Entries for "powers", "darken", "remove_3s" and "specification" for the table are illustrated in the following example. Asterisks and vertical bars are used as in Examples 37 and 38.

**Example 39:**

```
|
|/* SECTION III: EXTERNAL FUNCTIONS -- DESCRIPTIONS. ...*/
|...
|
|struct external_function_entry_t
|  external_functions_g[] =
|...
|  {
|
*|    {"powers",0,
*|      powers_,"F_POWERS_FUNCTION",
*|      "fortran_function",BY_REFERENCE,
*|
*|  "     () => LIST\
*| \n\
*| \n     Form a STAR LIST containing the powers of\
*| \ntwo from 1 to 1024."},
|
*|    {"darken",1,
*|      darken,"P_DARKEN_FUNCTION",
*|      "pascal_function",BY_VALUE,
*|
*|  "     (CONNECTION1) => CONNECTION\
*| \n\
*| \n     CONNECTION1 contains a pointer to a PASCAL\
*| \nrecord describing a color in terms of red, green\
*| \nand blue components.  Form a new CONNECTION\
*| \ncontaining a pointer to a related color which is\
*| \ndarkened to 70% of the original values."},
|
*|    {"remove_3s",1,
*|      remove_3s,"P_REMOVE_3S_FUNCTION",
*|      "pascal_function",BY_VALUE,
*|
*|  "     (LIST1) => LIST\
*| \n\
*| \n     Expects LIST1 to contain NUMBERs.  Removes\
*| \nall elements which are the NUMBER '3'.  Returns\
*| \nthe modified LIST."},
|
*|    {"specification",0,
*|      specification,"C_SPECIFICATION_FUNCTION",
*|      "function",BY_VALUE,
*|
```

```
* |"     () => LIST\
* |\n\
* |\n     Return a set of specifications for the next\
* |\nexperiment."},
  |
  |   {0,0,0,0,0,0,0} /* "End of table" entry. Do not remove. */
  | };
  |
```

Each entry contains seven components. A brief summary of these components is as follows.

**(1)** a reference name for the function within STAR,
**(2)** the number of arguments to the function,
**(3)** name of the function within its defining language,
**(4)** the label for the STAR CONNECTION to the function,
**(5)** the immediate parent class of the function,
**(6)** "BY_VALUE" or "BY_REFERENCE" argument passing, and
**(7)** comment text for the function within STAR.

Item 1, the name of the function within STAR, appears in lowercase, surrounded by double quotes, and conforms to the syntax of a STAR reference name. Item 2 is an integer constant ranging from 0 to 8. Due to constraints imposed by C, the number of arguments sent to an external function in STAR must be restricted to some preset upper limit. This has been chosen to be 8 arguments, although with a few modifications to the STAR interpreter code, the limit may be raised to a higher, finite number of arguments. Item 3 names the function as defined in its original language, and corresponds exactly to the name for the function appearing in SECTION II of the "starlink.c" file. As in SECTION II, some versions of FORTRAN will require a single underscore character to be appended to this name within C.

Item 4 provides a character string for labeling the CONNECTION placed in the "algorithm" field of the function's definition within STAR. This character string must contain only uppercase letters, digits, underscore characters and "@" characters. Each "@" character in the label string is replaced by a sequence of digits describing the integer value (address of the actual function in this case) stored within the CONNECTION. Item 5, the immediate parent class of the function, must be either "function" or one of the classes specified in Part I of the "starlink.c" file. Item 6 uses one of the constants "BY_VALUE" or "BY_REFERENCE" to indicate the method of passing arguments from STAR to the function, as determined by the compiler for its defining language. Typically, C and PASCAL functions take arguments by value while FORTRAN functions take arguments by reference. Some versions of PASCAL take arguments by reference, however, so it is important to check for a particular implementation. Finally, item 7 is a character string of the same form as the comment specification for classes of external functions, as

described for part I of "starlink.c". Specifically, it is a sequence of characters surrounded by double quotes and containing the sequence "\<cr>\n" for each carriage return to be included in the comment, where "<cr>" is the actual carriage return.

As with the definition of the table "external_function_classes_g" illustrated in Example 37, the table for individual external functions is ended with a special entry of all zeros ("{0,0,0,0,0,0,0}", above). This entry must not be removed, as the STAR interpreter uses the entry to detect the end of the table.

Given the above entries in "starlink.c" for the classes "fortran_function" and "pascal_function" along with the individual functions "powers", "darken", "remove_3s" and "specification", it may be useful to see the form of the definitions which would be created in STAR as a result. Suppose that the modifications described in Examples 37, 38 and 39 have been performed and the resultant "starlink.c" compiled and linked with the remaining STAR interpreter code plus code for the individual external functions. The following named RECORDs would appear in STAR, facilitating the calling of the functions "powers", "darken", "remove_3s" and "specification".

**Example 40:**

```
    fortran_function

{name -> FORTRAN_FUNCTION
 member_of -> class
 subclass_of -> function
 members -> [powers]
 subclasses -> []
 comment ->
    "    Functions defined in FORTRAN and linked
    "to the STAR interpreter."
 }

    pascal_function

{name -> PASCAL_FUNCTION
 member_of -> class
 subclass_of -> function
 members -> [darken remove_3s]
 subclasses -> []
 comment ->
    "    Functions defined in PASCAL and linked
    "to the STAR interpreter."
 }

    powers
```

```
{name -> POWERS
 member_of -> fortran_function
 comment ->
     "     () => LIST
     "
     "     Form a STAR LIST containing the powers of
     "two from 1 to 1024."
 n_arguments -> 0
 algorithm -> ^F_POWERS_FUNCTION
 }
```

darken

```
{name -> DARKEN
 member_of -> pascal_function
 comment ->
     "     (CONNECTION1) => CONNECTION
     "
     "     CONNECTION1 contains a pointer to a PASCAL
     "record describing a color in terms of red, green
     "and blue components. Form a new CONNECTION
     "containing a pointer to a related color which is
     "darkened to 70% of the original values."
 n_arguments -> 1
 algorithm -> ^P_DARKEN_FUNCTION
 }
```

remove_3s

```
{name -> REMOVE_3S
 member_of -> pascal_function
 comment ->
     "     (LIST1) => LIST
     "
     "     Expects LIST1 to contain NUMBERs. Removes
     "all elements which are the NUMBER '3'. Returns
     "the modified LIST."
 n_arguments -> 1
 algorithm -> ^P_REMOVE_3S_FUNCTION
 }
```

specification

```
{name -> SPECIFICATION
 member_of -> function
 comment ->
     "     () => LIST
     "
     "     Return a set of specifications for the next
     "experiment."
 n_arguments -> 0
 algorithm -> ^C_SPECIFICATION_FUNCTION
 }

    powers()                $ Sample call to "powers".

[1 2 4 8 16 32 64 128 256 512 1024]

    .a                      $ Suppose ".a" is a CONNECTION
                            $   specifying a color.
^COLOR_1003460

    darken(.a)              $ Sample call to "darken".

^COLOR_1254320

    remove_3s(              $ Sample call to "remove_3s".
     [1 3 4 6 3 2 3 3]
     )

[1 4 6 2]

    specification()         $ Sample call to
                            $   "specification".
[12 "calcite" [1 1 2]]
```

Following the modification of "starlink.c", compiling of the source files and linking of the generated object files into an executable version of STAR, the next step involves loading in symbolic definitions stored on text files and using the "suspend" function to create a second executable version of STAR additionally enhanced with these definitions. The "suspend" function operates in a fairly straightforward manner. Suppose that the above modifications have been made to "starlink" and that all object files concerned have been linked into an executable file named "star". The following example illustrates how the symbolic files "definitions_1" and "definitions_2" might be loaded into STAR and the resulting enhanced environment saved in a second executable file named "application". The example begins just after the file "star" has been invoked from the operating system level.

**Example 41:**

```
    load("definitions_1")

"definitions_1"

    load("definitions_2")

"definitions_2"

    suspend("application")

"application"

    exit()
```

Following the sequence above, the file "application" may be invoked from the operating system level, bypassing the need for loading in the definitions contained in "definitions_1" and "definitions_2" each time the application system is to be used.

One final aspect of constructing and executing the application system has been postponed until this point. When STAR is invoked from the operating system level, it is possible to specify, directly within the operating system call, one or more symbolic files to load into STAR and one or more zero-argument functions to call in STAR prior to the commencement of the interactive STAR evaluation environment. This mechanism is often employed when it is desired to take a user directly from the operating system level into the application system, thereby shielding him from the operation of STAR.

The general form for the "star" command is as follows. On the same line as the call to STAR ("star" in UNIX if the executable file produced by the linking process has been named as such), a sequence of the form "+ <file>" will cause the loading of a symbolic file "<file>", and a sequence of the form ": <function>" will cause the calling of the function "<function>", taking zero arguments. For example, the following (UNIX) sequence calls STAR from the operating system, loads the file "data" and calls the function "enter".

```
star + data : enter
```

For most application systems, the necessary symbolic files will have been loaded prior to this point and thus the invocation of the application system involves only a call to STAR coupled with a function call specification, such as the following:

```
application : begin
```

Here, "application" is the executable file created by "suspend" and containing both

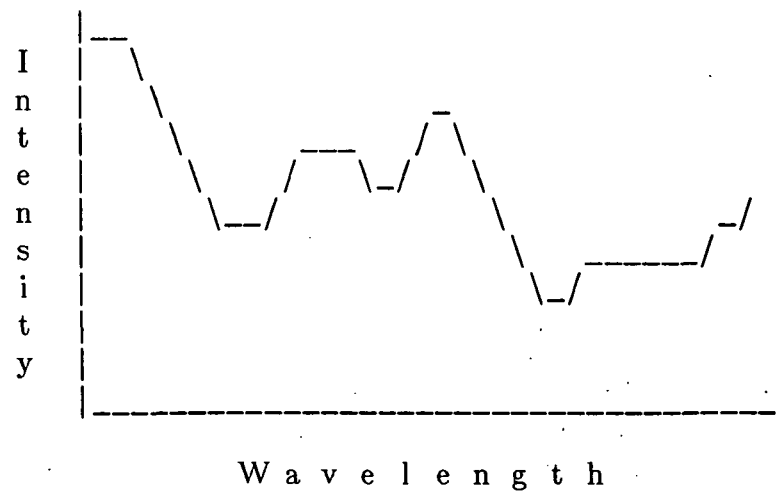
external function definitions and symbolic definitions. The function "begin" provides a top-level entrance to the application system operation.

## 19. A Sample Application

This section concludes the Tutorial Guide by presenting, in overview form, a sample application system involving STAR in the geological interpretation of imaging spectrometer data, a form of remote sensing data. The system, called SPECTRUM, is currently under development at the Jet Propulsion Laboratory and is constructed using a combination of routines and data structures defined in STAR and in the language C.

The imaging spectrometer is an instrument which measures intensity levels for a large number of contiguous bands in the electromagnetic spectrum in an imaging context. When mounted aboard an aircraft and transported over various sites of interest, the imaging spectrometer collects information which can be very useful in identifying surface materials such as rocks, minerals, water, vegetation and so forth. The instrument associated with the SPECTRUM system is the Airborne Imaging Spectrometer (AIS), which is carried aboard a C-130 aircraft operated by NASA and measures 128 bands in the near-infrared portion of the spectrum over an area approximately 300 meters wide and several kilometers in length on the ground. The resulting data set is three-dimensional, measuring 32 pixels wide by several hundred pixels in length and compounded over the 128 spectral bands. It is perhaps best to visualize this data set as a stack of 128 images taken of a scene, each image corresponding to the data for one particular wavelength of light.

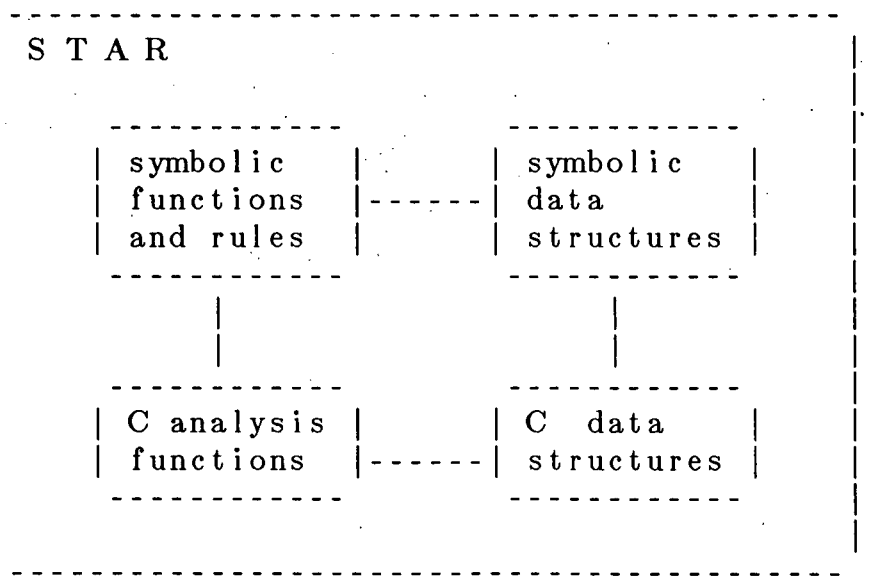
Viewing only a single pixel of a data set produced by the AIS instrument, one obtains a vector of 128 data values, one for each of the measured spectral bands. This vector may be plotted as a graph of intensity versus wavelength, as illustrated below.




A graph of intensity of light versus wavelength is often referred to as a "spectral signature" in this context. This is due to the fact that various materials, especially rocks and minerals, are known to exhibit distinguishable spectral reflectance characteristics and may often be identified in a laboratory setting largely from this data alone.

In the interpretation of Airborne Imaging Spectrometer data, a number of complicating factors reduce the accuracy by which materials may be identified. The Earth's atmosphere absorbs certain wavelengths of light to such an extent that these bands are not useful for remote sensing of surface materials. Also, since the size of a single pixel when projected to the ground is approximately 10 meters by 10 meters, it is often the case that several components combine to form the measured spectral curve. Perhaps the greatest obstacle, however, is simply the size of the data. For a typical AIS data set, there are approximately 2 million individual 8-bit data values. It is thus not practical to perform extensive analysis operations on the data set as a whole.

The approach taken in constructing the SPECTRUM system has been to combine a set of numerically oriented analysis routines defined in C with a top-level symbolic layer defined in STAR which manages the execution of these routines and the interpretation of results generated by the routines. In this manner, it is hoped that sufficiently accurate identifications of surface materials may be made without the expenditure of too great an amount of computational resources. The following diagram illustrates the basic components in the implementation of SPECTRUM.




Note in the above that there are two directions of coupling. In the horizontal direction, the STAR functions and rules operate primarily on symbolic structures in the STAR knowledge base and the C analysis functions on the C data structures. In the vertical direction, the STAR functions and rules make calls to the C analysis functions, and the STAR knowledge base contains CONNECTIONs to various C data structures.

The next few paragraphs outline briefly the individual structures combined to form the four major components listed above. The C data structures are considered first. The following is a list of the more important C data structures involved in the application:

(1) the original imaging spectrometer data set,
(2) a compressed version of this data set,
(3) a library of spectral signatures for minerals,
(4) other signatures generated in the analysis, and
(5) n-color maps of the image coverage area.

Elements listed under 3, 4 and 5 above are integrated into the STAR knowledge base using CONNECTIONs, as, for example,

^PLOT_1234400, ^PLOT_1716320, ^MAP_1006548,

where the memory addresses of the associated C data structures are included in the labels of the CONNECTIONs to distinguish these from one another.

The C analysis functions involved in the interpretation include the following:

(1) a function which extracts rough spectral features from the original data set, forming the compressed data set,

(2) functions for comparison of individual pairs of signatures by measures of varying complexity,

(3) a function which forms a map of the image coverage area indicating classes of pixels with related spectral reflectance characteristics,

(4) a function which decomposes a given signature into a linear combination of several other signatures, and

(5) functions for display of the image data and other graphics-oriented tasks.

Other C functions are included as well for performing specific operations on the C data structures. Taken together, the C functions and data structures form a general analysis framework for the imaging spectrometer data. Following is an illustration of one function listed under item 5 above, as it appears within the STAR environment.

```
{name -> DRAW_BAND
 member_of -> c_function
 comment ->
     "     (NUMBER1 NUMBER2 NUMBER3) => <boolean>
     "
     "     Display the NUMBER1'th band of the current
     "image, using NUMBER2 and NUMBER3 as the x and y
     "coordinates of the upper left hand corner for
     "the rectangle of display."
 n_arguments -> 3
 algorithm -> ^C_DRAW_BAND_FUNCTION
 }
```

Next to be considered is the STAR knowledge base, minus the functions and rules. A large portion of the STAR knowledge base in the SPECTRUM system is dedicated to the representation of a mineral classification hierarchy and associated data structures. Each included mineral classification is represented by a single named RECORD, describing a class of individual library signatures. For example, below is a listing of the class and two sample spectral signatures associated with the mineral muscovite in this knowledge base.

```
{name -> MUSCOVITE
 member_of -> class
 subclass_of -> mica
 members -> [muscovite_1 muscovite_2]
 subclasses -> []
 abundance -> very_common
 alteration_of -> [kyanite feldspar]
 alters_to -> []
 occurs_with -> [quartz albite orthoclase garnet]
 }

{name -> MUSCOVITE_1
 member_of -> muscovite
 data -> ^PLOT_958764
 }

{name -> MUSCOVITE_2
 member_of -> muscovite
 data -> ^PLOT_1501624
 }
```

Also contained within the STAR knowledge base are structures describing interpretative conclusions reached concerning the entire scene and the clusters of spectrally-related pixels.

Control in the SPECTRUM system is split between algorithmic (function evaluation) operation and rule-based operation. A top-level function maintains a dynamically-changing menu of options for the user of the system, allowing the user to proceed only along appropriate paths in the analysis. Also, the user is allowed to backtrack to previous points in the analysis to investigate alternative possibilities. For some steps in the analysis, the operation is carried out by user-defined functions in STAR which call particular C analysis functions. In other cases, the operation passes over to classes of rules, which accumulate interpretive conclusions and eventually return control to the top-level menu supervision function. The product of the analysis is a displayed map of the geographical region covered by the imaging spectrometer data, indicating areas exhibiting spectral similarity and likely surface minerals to be associated with these areas.

Some of the mechanics involved in piecing together the SPECTRUM system may carry over to other applications as well. For this application, it was found to be most convenient to create a single file directory containing the object files for all of the STAR interpreter code minus "starlink.c", which appears as both source and object code form, plus the source and object files for the C analysis functions and a set of text files defining the STAR structures, functions and rules. The list below illustrates the general distribution of files as they appear in the SPECTRUM application.

| interpreter obj. files | starlink files | C analysis functions | symbolic load files |
|---|---|---|---|
| starbifs.o | starlink.c | cluster.c | control.x |
| starcode.o | starlink.o | display.c | describe.x |
| starplus.o | | encode.c | display.x |
| starutil.o | | globals.c | explain.x |
| starini1.o | | graph.c | identify.x |
| starini2.o | | mixture.c | load.x |
| starini3.o | | ... | material.x |
| ... | | | |

Names ending in ".x" have been chosen for the text files defining STAR constructs. In addition to the above listed files, two other files, "LINK" and "SETUP.X" have been included to speed up reconstruction of the system after changes are made to any of the source files.

The file "LINK" specifies an operating system call to the linker and is used to form a new executable version of STAR whenever the C analysis functions are altered and recompiled. The file "LINK" may be executed directly from the operating system level in place of manually entering the link command and the files to be linked.

The other file, "SETUP.X", contains the STAR definition for a function "go" which makes repeated calls to "load": one for each text file to be loaded into the STAR environment. After the file "LINK" has been executed, forming a new version of the executable file named "star", the command

```
star + SETUP.X : go
```

enters the STAR environment and automatically loads in the symbolic definitions in the ".x" files. Once this is completed, it is only necessary to call "suspend" with the name of the final system ("spectrum" in this case), and a second executable file is created containing the STAR interpreter, the C analysis functions and data structures, and all symbolic definitions for the application. For SPECTRUM, the system is invoked from the operating system level by the following command

```
spectrum : begin
```

where "begin" is the name of a STAR function in the application environment which performs final initializations for an analysis session and starts up the application system.

## STAR Reference Manual

The STAR Reference Manual provides a concise summarization of many but not all of the features of the STAR language. It is intended for use as an aid in the development of STAR application systems, providing details on the various mechanisms and structures contained within the language which have been omitted for brevity in the Tutorial Guide. The Reference Manual is not intended as a substitute for the Tutorial Guide, however, and is thus written in a manner which facilitates consultation piece by piece but not sequential reading, start to finish. The reader may wish to glance through the Manual initially to grasp the basic structure of its organization and then refer to individual sections of the Manual as the need arises.

Some of the sections most relevant to the development of application systems are Section 3.3, Built-in Functions, which outlines the argument types, result types and operations of the STAR built-in functions, and Section 6, The STAR Utilities. Within the Tutorial Guide to STAR, the STAR utility functions are presented only in example form and simply listed in cases where several utilities perform analogous operations. Within Section 6 of the Reference Manual, the operation of each STAR utility is described in isolation.

Following the Reference Manual, the Appendix provides a list of names for all built-in structures in STAR. Argument and result types for built-in functions also appear in the Appendix.

# 1. General Operation of STAR

## 1.1 The Terminal Interface

This section describes the command syntax for calling STAR from the operating system level and the differences between evaluation of UNITs entered at the terminal and evaluation of UNITs elsewhere in STAR.

### Invoking the STAR interpreter

The STAR interpreter is invoked by the operating-system command

star <argument list>

where "<argument list>" is composed of argument pairs of the following two forms.

+ <filename>
: <function>

If the built-in STAR function "suspend" has been used to create an enhanced executable version of the STAR interpreter, the name generated in this operation is substituted for "star" in the above call. Following the call to STAR, the banner "STAR x.x" appears at the user terminal, and any operations specified in the argument list to STAR are executed.

In the command-line arguments to STAR, a space must occur between a "+" or ":" and the specified filename or function associated with that character. The two forms for arguments may be used interchangeably and in combination in any order. The operations specified by the arguments are carried out in the order in which the arguments appear. The "+" form specifies a text file to be loaded into STAR, as if the STAR built-in function "load" had been called for that file. The ":" form specifies a user-defined or built-in STAR function to be called. A function specified in this manner must take zero arguments.

### The evaluation loop

Following the processing of all command-line arguments to STAR, the interpreter enters into a cyclic process composed of three operations: (1) parsing a UNIT entered at the user terminal, (2) evaluating that UNIT and (3) displaying the UNIT resulting from evaluation. The cycle repeats until the function "exit" is called or the interpreter is halted through the use of escape characters defined in the host operating system.

### Indentation

Each time the interpreter enters the first phase of its cycle, parsing a new UNIT, it indents five spaces from the left margin. Whenever the user enters a carriage return within the new UNIT, the interpreter provides the original five spaces of indentation plus one additional space for each pending LIST, RECORD or EXPRESSION within the UNIT, and five additional spaces for each RECORD entry split between its attribute and value. STRINGs are also indented one space, printed as a double quote character for every line after the first.

All UNITs which are displayed by the interpreter are also indented in an analogous manner, starting flush with the left hand margin instead of five spaces to the right of this margin. When displaying, the interpreter splits a UNIT over several lines only when it would not be possible to print the entire UNIT on a single line. If a LIST containing only NUMBERs, TOKENs or named RECORDs will not fit on a single line, it is split up over several lines with each line containing as many elements as will fit. Other UNITs which will not fit on a single line are split up so that each element is listed on a separate line.

Automatic indentation of UNITs also occurs in the operation of the built-in STAR functions "parse" and "display" and in the storage of UNITs in text files by the built-in STAR functions "save" and "stash".

### Syntax checking

In a process associated with the indentation of UNITs entered at the user terminal, the interpreter also checks these UNITs for syntactic correctness. Syntactically invalid characters are detected upon entry of the next carriage return by the user. At such time as a syntactically invalid character is detected, the interpreter prints the message "SYNTAX ERROR: <c>" at the user terminal, where "<c>" is the offending character. The interpreter then starts a new line with the proper amount of indentation for the point at which the invalid character was entered and allows the user to proceed from this point as if the invalid character and all characters following it up to the end of its input line had never been entered.

The built-in functions "parse" and "unspell", which form a UNIT from input at the terminal or the contents of a STRING, respectively, may also generate syntax error messages similar to those incurred at the outer evaluation loop of STAR. The function "scan" behaves similar to "unspell" except that syntax errors are not announced; rather, the value "nil" is simply returned in such cases.

### Escape characters

In the entering of UNITs at the user terminal, the following four printable characters and two unprintable characters have special significance.

$ - Specifies in-line comments. When the interpreter encounters a "$" character on an input line, this characters and all characters following it up to but not including a carriage return character are discarded.

@ - Specifies preprocessing of a UNIT. The "@" character causes the next complete UNIT entered to be immediately evaluated following its parse. The evaluated result is then used in place of the parsed UNIT. (Note: use of "@" with input functions such as "parse" and "input" may produce unexpected results. This is because the input buffer may already contain one or more characters at the time the "parse" or "input" function is evaluated.)

% - Causes backtracking one level of nesting. If the "%" character is encountered between the elements of a LIST, between the entries or within an entry of a RECORD, or between the arguments of an EXPRESSION, the entire partially-formed LIST, RECORD or EXPRESSION is discarded. Several "%" characters may be entered in succession to backtrack more than a single level; however, after one or more adjacent "%" characters have been processed, the remainder of that input line is discarded. Thus, the user may not begin retyping the UNIT to be parsed until the next input line.

# - Retrieves the last UNIT returned by STAR. This UNIT is stored as the value of the built-in STAR variable "pound_sign" and is also accessible through the use of the "dot" function as in entering the EXPRESSION ".pound_sign". Note, however, that the EXPRESSION form of retrieving this UNIT requires evaluation for the actual retrieval of the desired UNIT, while entering the "#" character does not require evaluation.

**delete** - Causes the interpreter to backtrack a single character on the current input line.

**ctrl c** - Causes the present parse, evaluation or display to be halted and control returned to the top level cycle of the STAR interpreter, ready to parse the next UNIT. Depending upon the particular operating system, it may be necessary to follow a "ctrl c" character with a carriage return to realize the effect of this operation.

The four printable escape characters do not have special significance if entered within a STAR STRING. The two unprintable escape characters, however, retain their effect within STAR STRINGs. Other printable and unprintable escape characters may be defined by the operating system of the host computer.

## 1.2 Error Messages

The STAR interpreter classifies processing failures into the following six categories. All processing failures occur as a result of evaluating EXPRESSIONs.

**DEFINITION ERROR** - The function applied within the EXPRESSION or one of the variables listed as its arguments is improperly defined. Refer to the format for these definitions in Sections 4.3 and 4.4.

**APPLICATION ERROR** - The number of arguments to be passed to the function is not equal to the number specified in its definition.

**TYPE ERROR** - The indicated UNIT is not of the UNIT type (NUMBER, LIST, etc.) required for the specified argument of the function. Check the required UNIT type as specified in the Appendix.

**CLASS ERROR** - The indicated UNIT is a named RECORD as expected for the indicated argument of the function, but it is not a member of the correct class of named RECORDs (e.g., "attribute", "variable", etc.). Check the required named RECORD class as specified in the Appendix.

**VALUE ERROR** - The indicated UNIT is of the correct type and, if a named RECORD, the correct class; however, the particular value is out of range or otherwise unusable. Refer to the operation of the function indicated. The incorrectness of the specified value may be related in some way to the particular choice for other arguments to the function, or the value may be unusable independent of the other arguments.

**PERMISSION ERROR** - Evaluation of the EXPRESSION would have the effect of altering a portion of the initialized semantic network of STAR. As STAR is a highly transparent language (its internal workings are accessible), it is possible to specify operations which, if permitted, could corrupt the STAR operating environment. Refer to Section 3 for descriptions of the fixed language structures of STAR.

The type DEFINITION ERROR, above, occurs only for user-defined functions. An APPLICATION ERROR may occur for user-defined or built-in functions. The remaining errors occur only for built-in functions.

When a processing failure occurs in STAR, the interpreter prints a message at the user terminal indicating the error type and continues by displaying the UNIT primarily responsible for the failure, a partitioning line ("----------") and a trace listing of pending evaluations, in reverse order. The following codes appear in the trace listing, where "xxx" is a function name, "n" a positive integer value and "aaa" an attribute:

**(1)** xxx(n
**(2)** xxx(n*
**(3)** [n
**(4)** {aaa
**(5)** .....xxx(

The code "xxx(n" indicates that the failure occurred while evaluating the "n"'th argument of function "xxx", *prior* to the application of that function. The code "xxx(n*" indicates that application of the function "xxx" had commenced prior to the failure, and that the "n"'th argument was most closely related to the cause of the failure. The code "[n" indicates that the failure occurred while processing the "n"'th element of a LIST, presumably the body of a user-defined function or the body of a call to "do", "repeat", "while", "for" or "through". The code "{aaa" indicates that the failure occurred during the processing of the value of attribute "aaa" in a RECORD. Finally, the code ".....xxx(" indicates that the failure occurred during the application of the user-defined function "xxx".

The following is provided as an example of a STAR error message listing.

```
VALUE ERROR
0
----------
 /(2 [3 repeat(1* [4
.....f(
 set(2 [10
.....g(
```

The intepretation of the above sample is that the UNIT "0" was primarily responsible for the failure, comprising an improper value for the function "divide" (abbreviated "/"), and appearing as the second argument to "divide" ("/(2") within the third element ("[3") of the body of a "repeat" EXPRESSION ("repeat(1*", in process -- note the "*"), which appears as the fourth element ("[4") in the body of the user-defined function "f" (".....f("). In turn, "f" was called as a result of evaluating the second argument of an EXPRESSION involving the function "set" ("set(2"), this appearing as the tenth element ("[10") in the body of the function "g" (".....g("), which was called at the outermost level.

## 2. The STAR UNIT Types

This section describes the seven UNIT types within STAR, their syntax when entered or displayed at the user terminal, and evaluation rules applying to each.

### 2.1 NUMBERs

STAR NUMBERs correspond to single-precision floating point values in other languages. When entered at the user terminal, their syntax is as follows. A NUMBER may begin with a hyphen ("-") or digit, followed by zero or more additional digits, an optional decimal point, an optional set of digits after the decimal point, and an optional exponent specification. If included, the exponent specification must follow the preceding portions directly, with no intervening spaces or other delimiters. Such a specification takes the form "e" or "E" followed by an optional "+" or "-" followed by one or two digits.

Note that a single preceding minus sign ("-") is part of the syntax of a negative NUMBER when entered at the user terminal. When more than one hyphen precedes a digit, all but the last are taken to specify EXPRESSIONs involving the "negate" function. A hyphen which precedes an EXPRESSION is also taken to specify the "negate" function, while a hyphen which precedes a left parenthesis character ("(") is taken to specify the "subtract" function.

NUMBERs are displayed using formats similar to those for entry. For a NUMBER with no fractional component, the decimal point is omitted and the NUMBER takes on an "integer" appearance. Note that such NUMBERs are still stored in a floating point representation, however. For NUMBERs with sufficiently large exponents, the exponential format is used in display: for example, "2.34534e+11".

NUMBERs remain unaltered by evaluation.

### 2.2 TOKENs

TOKENs appear as "atomic" label values in STAR. As opposed to STRINGs, which exhibit a similar syntax, TOKENs are essentially indivisible values (unless converted into STRINGs with the built-in function "spell"). The syntax of a TOKEN is an initial uppercase letter followed by optional set of one or more uppercase letters, digits and underscore characters ("_"). TOKENs may be of any length; however, the characters must be sequential, and thus a carriage return at the end of a line will terminate the entry of a TOKEN.

TOKENs remain unaltered by evaluation.

## 2.3 STRINGs

STAR STRINGs are sequences of ASCII characters possibly containing carriage returns and thus comprising more than a single line. STRINGs are used for most textual quantities in STAR. The syntax of a STRING consists of a double quote character followed by a sequence of zero or more arbitrary characters and ending with a second double quote character.

When a STRING is entered at the user terminal, the interpreter indents one space for each line after the first; a double quote character is printed by the interpreter within this space of indentation and may be treated as a reminder that a STRING is pending. To enter a double quote as a character within a STRING, two consecutive double quote characters must be entered.

STRINGs are displayed much in the same manner as they are input. Initial and final double quote characters describe the extent of a STRING, and STRINGs consisting of more than a single line of text include a double quote character marking the beginning of each line after the first. Also, all double quotes contained as characters within a STRING are displayed as a pair of double quote characters. There is no ambiguity in the syntax of STRINGs as entered and displayed by these rules, and thus it is always possible to discriminate between double quote characters denoting the beginning and end of a STRING, those indicating succeeding lines of text and those indicating the inclusion of double quote characters within the text.

STRINGs remain unaltered by evaluation.

## 2.4 LISTs

LISTs describe sequences of zero or more UNITs in STAR. Depending on the application, the elements of a LIST may be interpreted as forming a sequence or merely forming an unordered set. In most cases, LISTs are used to describe homogeneous sets of UNITs, sharing some property of structure or function, while RECORDs are mostly used to describe heterogeneous sets of UNITs, where each UNIT has its own structure and function.

The syntax of a LIST is a left bracket character ("[") followed by zero or more UNITs, delimited by spaces, carriage returns, tabs or other non-printable characters, and ending with a right bracket character ("]"). As opposed to the list structure in languages like LISP and PROLOG, the LIST of STAR is itself a primitive structure and is not formed using a lower-level structure such as the "dotted pair". One consequence of this is that there is no "nil" value appearing at the end of a LIST.

LISTs are displayed either on a single line or distributed between several lines. For LISTs appearing on a single line, a single space character separates each element from the next element. For LISTs too large for a single line but containing only NUMBERs, TOKENs or named RECORDs, several lines are used, with each line containing as many elements as will fit, separated by space characters. For LISTs too large for a single line but containing elements other than these, a separate line is used for each element. Past 50 levels of nesting, a LIST is displayed simply as "[..]".

Evaluation of a LIST has no effect on the structure or any of the elements of the LIST. If it is desired to form a LIST containing the results of evaluating each element in a given LIST, it is possible to do this using the built-in STAR function "prepare".

## 2.5 RECORDs

RECORDs describe sets of associated pairs of UNITs in STAR. As opposed to LISTs, RECORDs most often represent heterogeneous combinations of structures. Each associated pair of UNITs is called an "entry" and is composed of an "attribute", which in most cases must be a named RECORD of the class "attribute", and a "value", which may be any type of UNIT.

The syntax of a RECORD is a left brace character ("{") followed by zero or more entries, delimited by spaces, carriage returns, tabs or other non-printable characters, and ending with a right brace character ("}"). The attribute and value of an individual RECORD entry are delimited by the two-character sequence hyphen-greater-than ("->"). The two characters of this delimiter must not be separated, although the delimiter as a whole may be separated from the preceding attribute and succeeding value by spaces, carriage returns, tabs or other non-printable characters.

A RECORD which contains an entry for the attribute "name" has special significance within STAR and is referred to as a "named RECORD". Named RECORDs usually contain entries as well for the attribute "member_of", specifying membership within classes of named RECORDs.

RECORDs are displayed in several formats. For an unnamed RECORD, if the entire RECORD will fit on a single line, it is printed as such, with the delimiter "->" placed directly between each attribute and value, with no intervening spaces. Each entry is delimited from the next by a single space character. For an unnamed RECORD which will not fit on a single line, a separate line is used for each entry, with the delimiter "->" plus an additional space before and after separating each attribute and value. For particular entries which do not themselves fit on a single line, the attribute is printed, followed by a space and the delimiter "->", and the value is printed on the succeeding line, indented an additional five spaces. Past 50 levels of nesting, an unnamed RECORD is displayed simply as "{..}".

Named RECORDs not displayed as elements within other UNITs are printed the same as unnamed RECORDs which occupy more than a single line. For a named RECORD which is displayed as an element within another UNIT, only the reference name is used.

Evaluation of a RECORD, whether named or unnamed, has no effect on the structure or any of the elements of the RECORD. If it is desired to form a RECORD containing the results of evaluating each attribute and value in an original RECORD, it is possible to do this using the built-in STAR function "prepare".

## 2.6 EXPRESSIONs

EXPRESSIONs specify commands to the STAR interpreter. The commands described by an EXPRESSION are carried out when the EXPRESSION is evaluated. There are three forms for the syntax of an EXPRESSION: a standard form and two abbreviated forms. In the standard form, the reference name for a STAR function is entered, followed immediately by a left parenthesis character ("("), followed by zero or more UNITs as arguments, and ending with a right parenthesis character (")"). The arguments may be delimited one from another by spaces, carriage returns, tabs or other non-printable characters.

The abbreviated forms for EXPRESSIONs may only be used for certain built-in STAR functions which have been initialized with an entry for the attribute "abbreviation". These are the functions "negate", "add", "subtract", "multiply", "divide", "dot", "enumerate", "equal", "less", "greater", "not", "and", "or" and "quote". For abbreviated EXPRESSIONs, if the function in question takes a single argument, the syntax consists of a single-character symbol specified by the "abbreviation" entry in the function's defining RECORD, followed immediately by the UNIT appearing as the argument to the function (e.g., ".a" or "~ true"). For abbreviated EXPRESSIONs involving functions which take two or more arguments (none of these functions take zero arguments), the syntax is the same as that for the standard form, except that the symbol designated for the function in question is substituted for the reference name of the function (e.g., "+(8 7)" or "&(true true)").

EXPRESSIONs are displayed much in the same manner as they are input. While it is possible to enter EXPRESSIONs in standard form for functions which may use the abbreviated forms, when these EXPRESSIONs are displayed, the abbreviated forms are always used. Past 50 levels of nesting, an EXPRESSION is displayed in standard form as "xxx(..)" or in abbreviated form as "s(..)" or "s..", where "xxx" is a reference name and "s" is a single-character abbreviation symbol.

When an EXPRESSION is evaluated, the arguments to the EXPRESSION are first evaluated. Next, the function specified in the EXPRESSION is applied, operating on the results of evaluating the arguments. Finally, the result returned by the applied function is returned for evaluation of the entire EXPRESSION. Built-in functions operate by executing compiled code defined in C. These functions are described individually in Section 3.3. The definition and application of user-defined functions is described in Section 4.3.

Normally, when an EXPRESSION is entered at the user terminal, its evaluation replaces that EXPRESSION with a UNIT returned by the applied function. The "quote" built-in function of STAR is used to spare an EXPRESSION from evaluation, preserving it for treatment as a data structure or for evaluation at a later time. EXPRESSIONs contained as elements within LISTs or RECORDs are also spared from evaluation as these structures are evaluated.

## 2.7 CONNECTIONs

CONNECTIONs are used in STAR to reference data structures and functions defined elsewhere in compiled programming languages. CONNECTIONs for external functions are normally set up during the initialization of the STAR interpreter as it is called from the operating system. These CONNECTIONs are formed according to specifications in the initialization tables for external functions located in the file "starlink.c". CONNECTIONs for external data structures are normally set up from within external functions by calling the STAR utility function "make_connection". CONNECTIONs formed in this manner may then be conveyed through the result value of the function into the STAR environment, or they may be inserted directly into the STAR knowledge base by the external functions, through calling various STAR utilities and built-in functions.

It is also possible to create a CONNECTION by typing it in at the user terminal. CONNECTIONs created in this manner, however, have only a label and no data value associated with them. Nevertheless, it is sometimes useful to create CONNECTIONs in this manner: once created they may be sent as arguments to external functions which insert the desired data values. When entered at the user keyboard, a CONNECTION has the following syntax. It begins with a circumflex character ("^"), followed by at least one and possibly more occurrences of uppercase letters, digits and the underscore character ("_").

CONNECTIONs are displayed using the same syntax as for entry. When a CONNECTION is created by a call to the utility "make_connection", a character string describing the label for the CONNECTION appears as one of the arguments of this call. This character string determines the displayed form for the newly created CONNECTION.

CONNECTIONs remain unaltered by evaluation.

## 3. Fixed Language Structures

This section describes each of the built-in named RECORDs in the initialized STAR knowledge base. These constructs constitute the fixed language structures of STAR, and the corresponding named RECORDs may not be altered in ways which would interfere with their specified function. The built-in RECORDs of STAR as presented in this section are listed by membership in the six major subclasses of the universal class "concept". These are the classes "class", "attribute", "function", "variable", "element" and "rule".

Much of the information presented in this section is extracted from the individual named RECORDs defined in STAR. The information for a particular construct, including its "comment" field may be accessed "on line" by entering the reference name for the construct.

### 3.1 Built-in Classes

#### "concept"

subclass of: (no parent class)
subclasses: class, attribute, function, variable, element, rule

The top or universal class. All named RECORDs are members of "concept". The six major subclasses of "concept" partition the set of named RECORDs according to primary function.

#### "class"

subclass of: concept
subclasses: (none as initialized)

The class of all classes. Classes are listed as members of "class" or its subclasses in addition to being organized into a hierarchy by the attributes "subclass_of" and "subclasses".

#### "attribute"

subclass of: concept
subclasses: (none as initialized)

The class of all attributes. Attributes are used as keys to the entries in RECORDs, appearing as the lefthand member of each entry pair. Automatic side-effects may be generated when a value for a particular attribute is asserted or retracted.

**"function"**

subclass of: concept
subclasses: (none as initialized)

The class of all functions. Contains as members all built-in functions of STAR. Also, external functions and user-defined functions may appear either as direct members of "function" or as members of user-defined subclasses of "function".

**"variable"**

subclass of: concept
subclasses: (none as initialized)

The class of all variables. Contains as direct members the built-in variables of STAR. User-defined variables may appear as direct members of "variable" or as members of user-defined subclasses of "variable".

**"element"**

subclass of: concept
subclasses: boolean, rule_mode

The class of all elements, or quantities in the domain of the application. The built-in element "nil" appears as a direct member of "element", while the remaining five built-in elements appear as members of the subclasses "boolean" and "rule_mode".

**"rule"**

subclass of: concept
subclasses: (none as initialized)

The class of all rules. Typically subdivided into a hierarchy of subclasses or rulesets. A class of rules is used as the first argument to "invoke".

**"boolean"**

subclass of: element
subclasses: (none as initialized)

The class of boolean values "true" and "false".

### "rule_mode"

subclass of: element
subclasses: (none as initialized)

The class of rule modes "single_test", "single_application" and "multiple_application".

## 3.2 Built-in Attributes

### Attributes used in defining all named RECORDs

#### "name"

The unique name given to a particular RECORD. Asserting or retracting values for "name" has automatic side-effects in the internal directory.

#### "member_of"

The immediate parent class of a named RECORD. Asserting or retracting values for "member_of" has automatic side-effects in the "members" entry for the parent class.

#### "comment"

Used to include comments within the definitions of named RECORDs of all classes.

### Attributes used in defining classes

#### "subclass_of"

The immediate encompassing class of a given class. Asserting or retracting values for "subclass_of" has automatic side-effects in the "subclasses" entry for the encompassing class.

**"members"**

The immediate members of a given class. Automatically updated by changes made to the "member_of" entries for individual named RECORDs.

**"subclasses"**

The immediate subclasses of a given class. Automatically updated by changes made to the "subclass_of" entries of other classes.

**"pattern"**

Used to store values and other aspects of attributes to be inherited from classes to members of classes.

**"aspects"**

An attribute used to distinguish a RECORD specifying various aspects of an attribute within a named RECORD, rather than simply the value for that attribute.

**"value"**

An aspect specifying the value of an attribute for a particular named RECORD when other aspects are present. The function "determine" is used to access an inherited value.

**"default"**

An aspect specifying a UNIT to be returned by the function "estimate" when applied to a particular named RECORD and attribute.

**"if_needed"**

An aspect specifying a UNIT to be evaluated and returned by the function "calculate" when applied to a particular named RECORD and attribute.

### "if_asserted"

An aspect specifying a LIST of functions to call whenever a value for a particular attribute is asserted.

### "if_retracted"

An aspect specifying a LIST of functions to call whenever a value for a particular attribute is retracted.

## Attributes used in defining attributes

### "side_effects"

The entry "side_effects -> true" within the definition of an attribute indicates that one or both of the aspects "if_asserted" and "if_retracted" are currently specified for that attribute.

## Attributes used in defining functions

### "abbreviation"

Used to specify a single character to appear in abbreviated EXPRESSIONs involving a particular built-in function. May not be used with user-defined functions.

### "n_arguments"

Used to specify the number of UNITs to be sent as arguments to a built-in function or other externally defined function.

### "arguments"

Specifies a LIST of variables to be used as arguments to a user-defined function.

### "temporary"

Specifies a LIST of variables to be given local bindings within the execution of a user-defined function.

### "algorithm"

Used to specify either a CONNECTION to an externally defined function or a LIST of UNITs defining the steps to be taken in the execution of a particular function.

## Attributes used in defining variables

### "bindings"

Used in the definition of a variable to specify a LIST of bindings. The first element in this LIST is the current value, accessible through the use of the "dot" function. Any other bindings are shadowed as a result of dynamic scoping rules.

## Attributes used in defining rules

### "mode"

Specifies the mode assigned to a rule as "single-test", "single-application" or "multiple-application".

### "condition"

Specifies the condition of a rule.

### "action"

Specifies the action of a rule. Also used in the context of the "branch" function to specify the action associated with a particular alternative.

## Attributes used for general programming

### "case"

Used in the context of the "branch" function to specify one of the possible alternatives for matching.

### "result_value"

Used by the function "result" to indicate a value to be returned by the "do" function.

### "break_value"

Used by the function "break" to indicate a value to be returned by "repeat", "while", "for" or "through".

### "skip_value"

Used by the function "skip" to indicate an immediate jump to the next iteration for the functions "repeat", "while", "for" or "through".

### "return_value"

Used by the function "return" to specify the value to be returned by a user-defined function.

### "stop_value"

Used by the function "stop" to specify an immediate halt to rule-based operation, plus a value to be returned by the "invoke" function.

## 3.3 Built-in Functions

In the descriptions of built-in functions in STAR appearing below, the following conventions have been used in the specification of argument and result types.

- To discriminate between arguments to a function which may share the same type specification, all argument specifications include a unique numerical designation (e.g., "NUMBER2", "<class1>", etc.). Numerical designations are not included in the specification of result types (e.g., "LIST", "<variable>").

- The word "UNIT" is used to specify arguments and results for which no type restrictions apply.

- Arguments and results which are restricted to one of the seven UNIT types are indicated using uppercase letters, as for example, "NUMBER1", "STRING", "LIST2", and so forth.

- Arguments and results which are restricted to be named RECORDs are specified using the name of the most general classification allowable, in lowercase and surrounded by angle brackets ("<" and ">"). Examples are "<concept1>", "<class>", "<attribute1>".

- Arguments which are *usually* sent via application of the "quote" function or previously spared from evaluation via the "quote" function are indicated with a preceding single quote character (e.g., "'UNIT1"). Note that this notation is intended to be merely a reminder and not a restriction placed upon the arguments.

### Functions for operations involving NUMBERs

#### "negate"

argument types: NUMBER1
result type: NUMBER
abbreviation: -

Arithmetic negation.

**"add"**

argument types: NUMBER1, NUMBER2
result type: NUMBER
abbreviation: +

Addition.

**"subtract"**

argument types: NUMBER1, NUMBER2
result type: NUMBER
abbreviation: -

Subtraction.

**"multiply"**

argument types: NUMBER1, NUMBER2
result type: NUMBER
abbreviation: *

Multiplication.

**"divide"**

argument types: NUMBER1, NUMBER2
result type: NUMBER
abbreviation: /

Division.

**"minimum"**

argument types: NUMBER1, NUMBER2
result type: NUMBER

Minimum of two NUMBERs.

**"maximum"**

argument types: NUMBER1, NUMBER2
result type: NUMBER

Maximum of two NUMBERs.

### Functions for operations involving TOKENs

**"locate"**

argument types: TOKEN1
result type: <concept>

Return the RECORD of name TOKEN1, creating one if necessary.

**"test"**

argument types: TOKEN1
result type: <concept>

Return the RECORD of name TOKEN1 if it exists; otherwise, return "nil".

### Functions for operations involving STRINGs

**"character"**

argument types: STRING1, NUMBER1
result type: STRING

Return a single-character STRING containing the NUMBER1'th character of STRING1. If NUMBER1 < 0, count from the end of STRING1.

**"fetch"**

argument types: STRING1, NUMBER1
result type: STRING

Return a STRING containing the first NUMBER1 characters of STRING1. If NUMBER1 < 0, use the last |NUMBER1| characters.

**"release"**

argument types: STRING1, NUMBER1
result type: STRING

Return a STRING containing all but the first NUMBER1 characters of STRING1. If NUMBER1 < 0, use all but the last |NUMBER1| characters.

**"join"**

argument types: STRING1, STRING2
result type: STRING

Return a STRING containing the concatenation of STRING1 and STRING2.

**"find"**

argument types: STRING1, STRING2
result type: NUMBER

Search for the first occurrence of STRING1 in STRING2, returning the number of characters before the occurrence if found, or the length of STRING2 if not found.

**"length"**

argument types: STRING1
result type: NUMBER

Return the number of characters in STRING1.

### Functions for operations involving LISTs

**"select"**

argument types: LIST1, NUMBER1
result type: UNIT

Return the NUMBER1'th element of LIST1. If NUMBER1 < 0, select the |NUMBER1|'th element from the end of LIST1.

**"replace"**

argument types: LIST1, NUMBER1, UNIT1
result type: LIST

Return a copy of LIST1 with the NUMBER1'th element replaced by UNIT1. If NUMBER1 < 0, replace the |NUMBER1|'th element from the end of LIST1.

**"delete"**

argument types: LIST1, NUMBER1
result type: LIST

Return a copy of LIST1 with the NUMBER1'th element deleted. If NUMBER1 < 0, delete the |NUMBER1|'th element from the end of LIST1.

**"insert"**

argument types: LIST1, NUMBER1, UNIT1
result type: LIST

Return a copy of LIST1 with UNIT1 inserted so that it occupies the NUMBER1'th position. If NUMBER1 < 0, insert at the |NUMBER1|'th position from the end of LIST1.

**"take"**

argument types: LIST1, NUMBER1
result type: LIST

Return a LIST containing the first NUMBER1 elements of LIST1. If NUMBER1 < 0, return a LIST containing the last |NUMBER1| elements of LIST1.

**"drop"**

argument types: LIST1, NUMBER1
result type: LIST

Return a LIST containing all but the first NUMBER1 elements of LIST1. If NUMBER1 < 0, return a LIST containing all but the last |NUMBER1| elements.

**"append"**

argument types: LIST1, LIST2
result type: LIST

Return a LIST containing the elements of LIST1 followed by the elements of LIST2.

**"size"**

argument types: LIST1
result type: NUMBER

Return the number of elements in LIST1.

**"union"**

argument types: LIST1, LIST2
result type: LIST

Treating LIST1 and LIST2 as sets, return a LIST containing the union of the elements in LIST1 and LIST2.

**"intersection"**

argument types: LIST1, LIST2
result type: LIST

Treating LIST1 and LIST2 as sets, return a LIST containing the intersection of the elements in LIST1 and LIST2.

**"difference"**

argument types: LIST1, LIST2
result type: LIST

Return a LIST corresponding to the set difference of the elements in LIST1 and the elements in LIST2.

## Functions for operations involving RECORDs

### "get"

argument types: RECORD1, <attribute1>
result type: UNIT

Return the value of <attribute1> in RECORD1.

### "put"

argument types: RECORD1, <attribute1>, UNIT1
result type: RECORD

Return a copy of RECORD1 with the value of <attribute1> set to be UNIT1, possibly replacing a previous value.

### "omit"

argument types: RECORD1, <attribute1>
result type: RECORD

Return a copy of RECORD1 with the entry for <attribute1> omitted.

### "detach"

argument types: RECORD1, LIST1
result type: RECORD

Return a RECORD containing in order the attributes found in LIST1 along with their corresponding values taken from RECORD1. Attributes contained in LIST1 but without values in RECORD1 are ignored.

### "attach"

argument types: RECORD1, RECORD2
result type: RECORD

Return a RECORD containing the attributes and values of RECORD1 followed by the attributes found in RECORD2 but not RECORD1, along with their corresponding values as specified in RECORD2.

**"key"**

argument types: RECORD1
result type: LIST

Return a LIST of the attributes in RECORD1.

**"image"**

argument types: RECORD1
result type: LIST

Return a LIST of the values specified for the attributes in RECORD1.

**"build"**

argument types: LIST1, LIST2
result type: RECORD

Return a RECORD formed by matching one-for-one the attributes in LIST1 with the values in LIST2.

**"define"**

argument types: RECORD1
result type: <concept>

Enter RECORD1 in the internal directory of named RECORDs and perform all side-effects resulting from asserting the specified values for the attributes in RECORD1. Return the resulting RECORD1.

**"create"**

argument types: <concept1>, <class1>
result type: <concept>

Assert that <concept1> is a member of <class1>, returning the new <concept1>.

**"assert"**

argument types: <concept1>, <attribute1>, UNIT1
result type: <concept>

Modify <concept1> to include the pair <attribute1> and UNIT1, performing all generated side-effects and returning the new <concept1> as the result. <attribute1> may not have a value in <concept1> at the start of this operation.

**"retract"**

argument types: <concept1>, <attribute1>
result type: <concept>

Modify <concept1> by removing <attribute1> and its value, if present, performing all generated side-effects and returning the new <concept1> as the result.

**"modify"**

argument types: <concept1>, <attribute1>, UNIT1
result type: <concept>

Modify <concept1> so that the value of <attribute1> is UNIT1, possibly replacing a previous value. Perform all side-effects generated by the retracting or asserting of values, returning the new <concept1> as the result.

**"revise"**

argument types: <concept1>, LIST1
result type: <concept>

Modify <concept1> so that it contains in order the attributes in LIST1 along with their values as specified in <concept1>, performing all generated side-effects and returning the new <concept1> as the result. Attributes in LIST1 but without values in the original <concept1> are ignored.

### "merge"

argument types: <concept1>, RECORD1
result type: <concept>

Modify <concept1> by including within it all attributes found in RECORD1 but not in <concept1>, along with their values as specified in RECORD1. Perform all side-effects generated by this process and return the new <concept1> as the result.

### "dot"

argument types: <variable1>
result type: UNIT
abbreviation: .

Return the current value of <variable1>, being the first element in the LIST given for the "bindings" attribute of <variable1>.

### "new"

argument types: <variable1>, UNIT1
result type: <variable>

Create a new binding for <variable1> by inserting UNIT1 as a new first element in the LIST specified for the "bindings" attribute of <variable1>, returning <variable1> as the result.

### "set"

argument types: <variable1>, UNIT1
result type: <variable>

Modify the current value of <variable1> by replacing the first element of the LIST given for the "bindings" attribute of <variable1> with UNIT1. Return <variable1> as the result.

### "old"

argument types: <variable1>
result type: <variable>

Remove the current top binding of <variable1>, specified as the first element in the LIST given for the "bindings" attribute of <variable1>, returning <variable1> as the result.

**"determine"**

argument types: <concept1>, <attribute1>
result type: UNIT

Return the value of <attribute1> in <concept1>. If none exists, use an inherited value for <attribute1> as found in a class containing <concept1>, starting with its immediate parent class.

**"estimate"**

argument types: <concept1>, <attribute1>
result type: UNIT

Return an estimated value of <attribute1> for <concept1>, as found in the "default" aspect of <attribute1> in <concept1>. If none exists, use an inherited default aspect as found in a class containing <concept1>, starting with its immediate parent class.

**"calculate"**

argument types: <concept1>, <attribute1>
result type: UNIT

Return a calculated value of <attribute1> for <concept1>, as found by evaluating the "if_needed" aspect of <attribute1> in <concept1>. If none exists, use an inherited "if_needed" aspect as found in a class containing <concept1>, starting with its immediate parent class.

**"obtain"**

argument types: <concept1>, <attribute1>, <attribute2>
result type: UNIT

Return the <attribute2> aspect of <attribute1> as found in <concept1>. If none exists, use an inherited aspect, as found in a class containing <concept1>, starting with its immediate parent class.

### "path"

argument types: <concept1>
result type: LIST

Return a LIST with first element "concept", last element <concept1>, and intermediate elements specifying the chain of subclasses leading from "concept" to <concept1>.

### "enumerate"

argument types: <class1>
result type: LIST
abbreviation: :

Return a LIST containing all members of <class1> or any of its subclasses.

## Functions for operations involving EXPRESSIONs

### "operation"

argument types: 'EXPRESSION1
result type: <function>

Return the operating function within EXPRESSION1.

### "application"

argument types: 'EXPRESSION1
result type: LIST

Return a LIST of the unevaluated arguments contained within EXPRESSION1.

### "formulate"

argument types: <function1>, LIST1
result type: EXPRESSION

Construct and return an EXPRESSION in which <function1> operates on the arguments specified in LIST1.

## Functions for logical operations

### "number"

argument types: UNIT1
result type: <boolean>

Return "true" if UNIT1 is a NUMBER; otherwise return "false".

### "token"

argument types: UNIT1
result type: <boolean>

Return "true" if UNIT1 is a TOKEN; otherwise return "false".

### "string"

argument types: UNIT1
result type: <boolean>

Return "true" if UNIT1 is a STRING; otherwise return "false".

### "list"

argument types: UNIT1
result type: <boolean>

Return "true" if UNIT1 is a LIST; otherwise return "false".

### "record"

argument types: UNIT1
result type: <boolean>

Return "true" if UNIT1 is a RECORD; otherwise return "false".

**"expression"**

argument types: UNIT1
result type: <boolean>

Return "true" if UNIT1 is an EXPRESSION; otherwise return "false".

**"connection"**

argument types: UNIT1
result type: <boolean>

Return "true" if UNIT1 is a CONNECTION; otherwise return "false".

**"null"**

argument types: UNIT1
result type: <boolean>

Return "true" if UNIT1 is the named RECORD "nil"; otherwise return "false".

**"equal"**

argument types: UNIT1, UNIT2
result type: <boolean>
abbreviation: =

Return "true" or "false" based on equivalence of memory address for named RECORDs or CONNECTIONs and displayed form for other types of UNITs.

**"less"**

argument types: NUMBER1, NUMBER2
result type: <boolean>
abbreviation: <

Return "true" if NUMBER1 is less than NUMBER2; otherwise return "false".

**"greater"**

argument types: NUMBER1, NUMBER2
result type: <boolean>
abbreviation: >

Return "true" if NUMBER1 is greater than NUMBER2; otherwise return "false".

**"in"**

argument types: UNIT1, LIST1
result type: <boolean>

Return "true" if UNIT1 is contained in LIST1; otherwise return "false".

**"subset"**

argument types: LIST1, LIST2
result type: <boolean>

Return "true" if the elements of LIST1 form a subset of the elements in LIST2; otherwise return "false".

**"isa"**

argument types: <concept1>, <class1>
result type: <boolean>

Return "true" if <concept1> is a member of <class1> or of a subclass of <class1>; otherwise return "false".

**"within"**

argument types: <class1>, <class2>
result type: <boolean>

Return "true" if <class1> is a subclass of <class2> or of any subclasses of <class2>; otherwise return "false".

**"not"**

argument types: <boolean1>
result type: <boolean>
abbreviation: ~

Logical negation.

**"and"**

argument types: <boolean1>, <boolean2>
result type: <boolean>
abbreviation: &

Logical conjunction of two boolean values.

**"or"**

argument types: <boolean1>, <boolean2>
result type: <boolean>
abbreviation: |

Logical disjunction of two boolean values.

**"exists"**

argument types: LIST1, <variable1>, 'UNIT1
result type: <boolean>

Return "true" if there is an element of LIST1 for which UNIT1 evaluates to "true" when <variable1> is bound to that element. Otherwise return "false".

**"every"**

argument types: LIST1, <variable1>, 'UNIT1
result type: <boolean>

Return "true" if for each element of LIST1, evaluation of UNIT1 returns "true" when <variable1> is bound to that element. Otherwise return "false".

**"which"**

argument types: LIST1, <variable1>, 'UNIT1
result type: LIST

Return a LIST containing each element of LIST1 for which UNIT1 evaluates to "true" when <variable1> is bound to that element.

## Input/output and translation functions

**"parse"**

argument types: (no arguments)
result type: UNIT

Parse input from the terminal and create a UNIT, returning that UNIT as the result.

**"display"**

argument types: UNIT1
result type: UNIT

Display UNIT1 at the terminal, returning UNIT1 as the result.

**"input"**

argument types: (no arguments)
result type: STRING

Read in a line of characters from the terminal, forming a STRING containing those characters (including the final carriage return) and return that STRING as the result.

**"output"**

argument types: STRING1
result type: STRING

Write the contents of STRING1 to the terminal and return STRING1 as the result.

**"format"**

argument types: STRING1, LIST1
result type: STRING

Copies STRING1, replacing all occurrences of the circumflex ("^") character followed by a positive integer specification with the spelled version of the indicated element in LIST1, following evaluation of that element. Double quotes are omitted in STRINGs converted from the LIST. Two adjacent circumflex characters are translated to a single circumflex character in the resultant STRING, with no conversion taking place.

**"save"**

argument types: STRING1
result type: STRING

Save in symbolic form all changes made to the initialized knowledge base, using the contents of STRING1 as a path/filename. Return STRING1 as the result.

**"stash"**

argument types: STRING1, LIST1
result type: STRING

Save in symbolic form the definitions of the named RECORDs contained in LIST1, using the contents of STRING1 as a path/filename. Return STRING1 as the result.

**"load"**

argument types: STRING1
result type: STRING

Parse and evaluate the UNITs contained in symbolic form in the file specified by STRING1.

**"read"**

argument types: STRING1
result type: STRING

Read the contents of the text file specified by STRING1 and form a STRING containing those characters. Return this STRING as the result.

**"write"**

argument types: STRING1, STRING2
result type: STRING

Create or rewrite the file specified by STRING1, using the characters of STRING2 as text. Return STRING1 as the result.

**"extend"**

argument types: STRING1, STRING2
result type: STRING

Create or append to the file specified by STRING1, using the characters of STRING2 as text. Return STRING1 as the result.

**"spell"**

argument types: UNIT1
result type: STRING

Construct and return a STRING corresponding to the characters in the displayed form of UNIT1.

**"unspell"**

argument types: STRING1
result type: UNIT

Parse STRING1 for the specification of a UNIT, returning that UNIT as the result.

**"scan"**

argument types: STRING1
result type: UNIT

Parse STRING1 for the specification of a UNIT, returning that UNIT as the result. Differs from "unspell" in that errors are not announced; rather, the value "nil" is simply returned in such cases.

## Functions related to programming

### "quote"

argument types: UNIT1
result type: UNIT
abbreviation: '

Spare UNIT1 from evaluation, returning it in its unevaluated form as the result.

### "evaluate"

argument types: 'UNIT1
result type: UNIT

Evaluation of a UNIT.

### "prepare"

argument types: UNIT1
result type: UNIT

Evaluation of a UNIT with the difference that if UNIT1 is a RECORD or LIST, all UNITs contained as elements of that RECORD or LIST are also evaluated.

### "apply"

argument types: <function1>, LIST1
result type: UNIT

Application of <function1> to the elements of LIST1 as arguments.

### "if"

argument types: <boolean1>, 'UNIT1
result type: UNIT

If <boolean1> is "true", evaluate UNIT1 and return the result of that evaluation. Otherwise return "nil".

### "ifelse"

argument types: <boolean1>, 'UNIT1, 'UNIT2
result type: UNIT

If <boolean1> is "true", evaluate UNIT1 and return the result of that evaluation. Otherwise evaluate UNIT2 and return the result of that evaluation.

### "branch"

argument types: UNIT1, LIST1
result type: UNIT

Search the RECORDs which are the elements of LIST1 for one with a value for the attribute "case" which evaluates to a UNIT equal to UNIT1. If such a RECORD is found, evaluate the value of the "action" attribute for that RECORD and return the result of evaluation.

### "do"

argument types: LIST1
result type: UNIT

Sequentially evaluate the elements of LIST1, interrupting the process if the result of evaluating an element is a RECORD with a value for the attribute "result_value". In this case, return the value of "result_value" for that RECORD as the result of "do". Otherwise, return "nil" following evaluation of the last element in LIST1.

### "repeat"

argument types: LIST1
result type: UNIT

Repeatedly evaluate the elements of LIST1 in a sequential manner, skipping to the next iteration or exiting the looping process if the evaluation of an element results in a RECORD with "skip_value" or "break_value" attribute, respectively. In the latter case, return the value of "break_value" in the RECORD as the result for "repeat".

### "while"

argument types: 'UNIT1, LIST1
result type: UNIT

Repeatedly evaluate the elements of LIST1 in a sequential manner as long as UNIT1 evaluates to "true" prior to each iteration. Skip to the next iteration or exit the loop in the same manner as for the function "repeat".

### "for"

argument types: 'UNIT1, 'UNIT2, 'UNIT3, LIST1
result type: UNIT

Evaluate UNIT1 and continue by repeatedly carrying out a cyclic process of evaluating UNIT2, sequentially evaluating the elements of LIST1, then evaluating UNIT3. The evaluation of UNIT2 is expected to produce a boolean value. If this value is not "true" for a particular iteration, exit the looping process directly, returning the value "nil". Also, skip to the next iteration or exit the loop in the same manner as for the function "repeat".

### "through"

argument types: LIST1, <variable1>, LIST2
result type: UNIT

Repeatedly evaluate the elements of LIST2 in a sequential manner, one iteration for each element of LIST1. Before each iteration, bind <variable1> to the current element of LIST1. Skip to the next iteration or exit the loop in the same manner as for the function "repeat".

### "invoke"

argument types: <class1>, LIST1
result type: UNIT

Invoke rule-based operation, using the members of <class1> as candidate rules. LIST1 specifies the values of the arguments to the ruleset, similar to the arguments to a function. Rule-based operation continues until no rules fire on a given cycle, the list of candidate rules becomes empty or the result of applying a particular rule is a RECORD with attribute "stop_value" (resulting from use of the "stop" function). In this case, return the value associated with "stop_value" as the result. Uses the built-in variables "control" and "alternatives".

**"result"**

argument types: UNIT1
result type: RECORD

Return a RECORD containing a single attribute, "result_value", with UNIT1 as its value. Used in conjunction with the "do" function.

**"break"**

argument types: UNIT1
result type: RECORD

Return a RECORD containing a single attribute, "break_value", with UNIT1 as its value. Used in conjunction with the functions "repeat", "while", "for" and "through".

**"skip"**

argument types: (no arguments)
result type: RECORD

Return a RECORD containing a single attribute, "skip_value", with "nil" as its value. Used in conjunction with the functions "repeat", "while", "for" and "through".

**"return"**

argument types: UNIT1
result type: RECORD

Return a RECORD containing a single attribute, "return_value", with UNIT1 as its value. Used to return a resulting value from a user-defined function.

**"stop"**

argument types: UNIT1
result type: RECORD

Return a RECORD containing a single attribute, "stop_value", with UNIT1 as its value. Used in conjunction with the function "invoke".

### Miscellaneous functions

#### "pause"

argument types: STRING1
result type: STRING

Often called from within a user-defined function for diagnostic purposes. STRING1 is printed at the user terminal, minus the surrounding quotes, then a series of UNITs may be entered at the terminal to be evaluated and the results displayed. The process terminates when the value "nil" is entered. STRING1 is returned as the result.

#### "system"

argument types: STRING1
result type: STRING

Use the characters of STRING1 to specify a command to the operating system. Returns STRING1 as the result.

#### "suspend"

argument types: STRING1
result type: STRING

Saves an image of the currently running executable file in a file named by STRING1, so that the session may be reentered at a later time. STRING1 is returned.

#### "exit"

argument types: (no arguments)
result type: (permanent exit from STAR)

Exit the STAR interpreter.

## 3.4 Built-in Variables

#### "pound_sign"

Used to store the last UNIT returned by the STAR interpreter in the user cycle. The value may be retrieved as ".pound_sign" or by using the special escape character "#".

**"control"**

Used to store the current ruleset in use by the function "invoke". The value may be accessed but not altered by the user.

**"alternatives"**

Used to store the currently active members of the ruleset in use by "invoke". The value may be accessed but not altered by the user.

## 3.5 Built-in Elements

**"nil"**

parent class: element

Flag value used to indicate the absence of an otherwise meaningful value.

**"true"**

parent class: boolean

The logical value "true".

**"false"**

parent class: boolean

The logical value "false".

**"single_test"**

parent class: rule_mode

Specifies that a rule is to be tested only once, possibly applied, and then removed from the LIST of active rules.

**"single_application"**

parent class: rule_mode

Specifies that a rule may be tested any number of times, but must be removed from the LIST of active rules directly following its first application.

**"multiple_application"**

parent class: rule_mode

Specifies that a rule may be tested and applied any number of times without being removed from the LIST of active rules.

## 3.6 Built-in Rules

The STAR knowledge base as initialized does not contain named RECORDs listed under the class "rule".

## 4. Application-dependent Structures

This section describes the format and function of user-defined named RECORDs in STAR. These named RECORDs appear alongside the built-in named RECORDs in the STAR knowledge base and tailor the STAR environment for the development and operation of specific application systems. In a similar manner to that of Section 3, this section is divided into six subsections, corresponding to the six major subclasses of the universal class "concept".

### 4.1 User-defined Classes

| REQUIRED ENTRIES: | OPTIONAL ENTRIES: |
|---|---|
| "name" | "members" |
| "member_of" | "subclasses" |
| "subclass_of" | "comment" |
| | (others as needed) |

User-defined classes have the same general form as that of built-in classes. Individual user-defined classes may be designated either as direct members of the class "class" or members of user-defined subclasses of the class "class".

The attribute "subclass_of" specifies a class which is to be the immediate encompassing class or superset of the class in question. Note the difference between the parent class (value of "member_of"), which contains a given class as an actual *member* (the parent class in this case is thus a "class of classes") and the immediate superset class, which generalizes the members of the given class to include other members listed explicitly or drawn from other subclasses.

The attributes "members" and "subclasses" provide reciprocal listings from parent classes and superset classes to their individual members and subclasses. These entries are considered "optional" but are actually required if the class is to contain individual members or subclasses. Normally, entries for both "members" and "subclasses" are omitted in the initial definition of a class, to be set up and updated automatically by the built-in side-effect mechanism as individual members and subclasses are defined.

Classes are usually defined in a top-down fashion, from parent classes and superset classes to their individual members and subclasses. This ensures the presence of "members" and "subclasses" entries in parent classes and superset classes to which modifications are made when individual members and subclasses are defined. While it is permitted to modify the "members" and "subclasses" entries for user-defined classes directly (this is not permitted for built-in classes), it is not recommended to rely upon this manner of operation as it is easy to introduce inconsistencies into the framework of the STAR knowledge base.

The attribute "comment" is usually associated with a STRING describing the general structure and function of the class.

## 4.2 User-defined Attributes

| REQUIRED ENTRIES: | OPTIONAL ENTRIES: |
|---|---|
| "name" | "comment" |
| "member_of" | "side_effects" |
| | (others as needed) |

User-defined attributes also have the same general form as that of their built-in counterparts. Individual user-defined attributes may be designated either as direct members of the class "attribute" or members of user-defined subclasses of the class "attribute".

The attribute "side_effects" specifies whether or not the STAR interpreter is to search for side-effect specifications whenever entries for the given attribute are added, changed or removed in any named RECORD. If an entry for "side_effects" is not given, the value is assumed to be "false". Otherwise, the value of "side_effects" is specified as either "true" or "false". The specification of operations to perform as side-effects is described in Section 10 of the Tutorial Guide.

Two other mechanisms related to attributes involve the inheritance of values by members of classes and the specification of "aspects" of an attribute other than the value. These mechanisms are also described in Section 10 of the Tutorial Guide.

In the context of attributes, the "comment" entry may be used to include a STRING describing in English the range of values acceptable for the given attribute, various aspects which are used for that attribute, and the operations involved in any side-effects for the attribute.

## 4.3 User-defined Functions

| REQUIRED ENTRIES: | OPTIONAL ENTRIES: |
|---|---|
| "name" | "comment" |
| "member_of" | "arguments" |
| "algorithm" | "temporary" |
| | (others as needed) |

The form for user-defined functions is somewhat different than the form for built-in and external functions in STAR. This stems from the difference in operation between the types of functions. Built-in and external functions operate by executing compiled code linked to the STAR interpreter through the use of CONNECTIONs. User-defined functions operate by sequentially evaluating the elements of a STAR LIST, usually containing EXPRESSIONs involving actions to be taken by STAR. User-defined functions may appear either as direct members of the class "function" or as members of user-defined

subclasses of "function".

For a user-defined function, the "algorithm" attribute is associated with a LIST of UNITs to be evaluated in a sequential manner. After setting up new bindings for a function call, the interpreter proceeds by evaluating each element in this LIST, one after another. If the resulting value of evaluating any element is a RECORD with an entry for "return_value" (resulting from a call to "return"), the process is halted and the value of "return_value" returned for the function call. In this manner, it is possible both to exit from the function execution before normal completion and to return specific values as results for the function call.

The attributes "arguments" and "temporary", if present, are associated with LISTs containing zero or more variables. When a user-defined function is called, the argument variables are given new bindings corresponding to the evaluated arguments to the function. Also at this time, the variables in the "temporary" LIST are given new bindings initialized to the value "nil". If "bindings" entries are absent in the definitions for particular variables, these are set up prior to inserting the new binding. Upon completion of the function call, the new bindings for both the argument variables and temporary variables are removed. If there are no arguments to a user-defined function, this may be indicated either by an entry associating "arguments" with an empty LIST or by the absence of an entry for "arguments". Likewise, if no variables are to be given local bindings, an entry associating "temporary" with an empty LIST may appear, or "temporary" entry itself may be absent.

The "comment" attribute may be used to specify the domain and range of the function as well as the general operation of the function. A consistent format has been used for this entry in the definitions of STAR built-in functions; the programmer may wish to borrow this format for use with user-defined functions as well.

## 4.4 User-defined Variables

| REQUIRED ENTRIES: | OPTIONAL ENTRIES: |
|---|---|
| "name" | "comment" |
| "member_of" | "bindings" |
| | (others as needed) |

User-defined variables parallel the structure of built-in variables in STAR. Individual user-defined variables may appear either as direct members of the class "variable" or as members of user-defined subclasses of "variable".

The attribute "bindings", while technically optional, is necessary for a variable to be assigned a value. An entry for "bindings" is automatically set up when a new binding is created, either by the built-in function "new" or by the application of a user-defined function which lists a variable under "arguments" or "temporary".

The value of "bindings" in the definition of a variable is a LIST of UNITs, the individual bindings for that variable. The first element in this LIST is the current binding and is retrieved through application of the "dot" function to the variable. If a variable appears as an argument or a temporary variable to a user-defined function, new bindings are created and destroyed automatically with the application of these functions. It is also possible to modify the bindings of any variable directly through the use of the built-in functions "new", "set" and "old", or, alternatively, through use of the more general-purpose functions such as "assert" and "modify".

For a variable, the "comment" entry may be used to specify the range of acceptable values for bindings, functions which use the variable either as an argument or temporary value, and functions which treat the variable as an external "global" value.

## 4.5 User-defined Elements

| REQUIRED ENTRIES: | OPTIONAL ENTRIES: |
|---|---|
| "name" | "comment" |
| "member_of" | (others as needed) |

The form for user-defined elements is quite flexible and depends upon the organization of a particular application system. As the class "element" is intended to include named RECORDs specifying application-dependent quantities, this class offers a "catch-all" environment for structures which do not fit into the other five major subdivisions of "concept". Individual user-defined elements may appear as direct members of the class "element" but most often appear as members of user-defined subclasses of "element".

The "comment" entry for elements may be used for documentation as needed according to the nature of the application.

## 4.6 User-defined Rules

### Individual rules

| REQUIRED ENTRIES: | OPTIONAL ENTRIES: |
|---|---|
| "name" | "comment" |
| "member_of" | (others as needed) |
| "mode" | |
| "condition" | |
| "action" | |

As the built-in function "invoke" operates upon a class of rules, it is often the case that the class "rule" is subdivided into several user-defined subclasses, each of which contains several individual rules as members. Only when there is just a single set of rules which may be invoked within an application system will the rules appear as direct members of the class "rule" (in this case, "rule" itself is used as the first argument when calling "invoke").

Within the definition for an individual rule, the attribute "mode" is associated with one of the three rule modes "single_test", "single_application" and "multiple_application". In the invocation of a given ruleset, rules designated "single_test" are tested only a single time (and possibly fired as a result). Rules designated "single_application" may be tested several times but only fired a single time. Finally, rules designated "multiple_application" may be tested and fired an unlimited number of times.

The attributes "condition" and "action" specify, respectively, the test for circumstances under which a rule is to fire and the operation to take place in firing the rule when its condition evaluates to "true". The "comment" attribute may be used to document the relationship of the given rule to other rules and to particular data structures which may be affected by firing of the rule. Other entries which may commonly be included in the definitions of individual rules are those which provide data structures to be used when tracing rule-firing history lists for the generation of explanations.

**Classes of rules**

| REQUIRED ENTRIES: | OPTIONAL ENTRIES: |
|---|---|
| "name" | "members" |
| "member_of" | "subclasses" |
| "subclass_of" | "comment" |
| | "arguments" |
| | "temporary" |
| | (others as needed) |

Classes of rules have their own particular structure in STAR. The "member_of", "subclass_of", "members" and "subclasses" entries operate in the same manner as for other classes. The "arguments" and "temporary" entries operate in a manner similar to the entries for these attributes in user-defined functions.

When a class of rules is invoked, the arguments as specified in the call to "invoke" are evaluated and passed along to the class of rules, where they are stored as new bindings of the variables listed under "arguments". Also at this time, the variables listed under "temporary" are given new bindings initialized to "nil", parallel to the operation of user-defined functions. The interpreter then enters a rule-based operation mode, executing the members of the class of rules as a production system. Upon completion, a return value (if specified by a call to the function "stop") is passed back for the return value from "invoke", and the current bindings are removed for the argument and temporary variables of the class of rules.

The "comment" entry for a class of rules may be used to document the origins of calls to "invoke" for the class, other classes of rules which may be invoked from within the operation of the given class, and the general function of the class within the framework of the application system as a whole.

## 5. Assembling Application Systems

This section summarizes the process by which the application programmer forms a complete system from component source files. The process is partitioned into six sequential steps, covered, respectively, in Sections 5.1 through 5.6.

### 5.1 Defining External Functions

This section provides checklists of items related to the definition and declaration of external functions to be linked with the STAR interpreter.

**Within the compiled language source files**

**(1)** Each compiled language source file should contain, if possible, a type declaration equating the type "unit" with the type "integer". If present, this declaration should appear at the top of the file. The following samples illustrate the declaration in C and in PASCAL.

```
typedef int unit;          (in C)
type unit = integer;       (in PASCAL)
```

**(2)** If the compiled language passes arguments by value, only the "by value" STAR utility functions should be called from within external functions. Likewise, if the language passes arguments by reference, only the "by reference" utilities should be used. Also, if the external function calls built-in functions of STAR, only the corresponding "by value" or "by reference" versions of these functions should be used. C and some versions of PASCAL pass arguments by value, while FORTRAN and other versions of PASCAL pass arguments by reference. If the programmer is unsure, the correct mechanism may often be determined simply by trial and error.

**(3)** Care should be taken to declare the STAR utilities and built-in functions used by the external functions correctly within the compiled language source files. Data types for the arguments and results of the utilities are provided in Section 6 of the Reference Manual. All STAR built-in functions take zero or more UNITs as arguments and return a UNIT as their result.

### Within the file "starlink.c"

(1) Each class of external functions should be described by a single entry in the table "external_function_classes_g" appearing in the area marked "SECTION I: ..." of the file "starlink.c". Section 18 of the Tutorial Guide describes the format for these entries.

(2) An external declaration for each of the external functions should appear in the area marked "SECTION II: ..." of "starlink.c". These declarations take the form

```
extern struct unit_t *xxx();
```

where "xxx" is the name of an external function as it appears in its defining language. (Some versions of FORTRAN require an underscore character to be appended to this name in "starlink.c": for example, "xxx_" in the above declaration.)

(3) Each external function should be described by a single entry in the table "external_functions_g" appearing in the area marked "SECTION III: ..." of "starlink.c". The format for these entries is also described in Section 18 of the Tutorial Guide.

## 5.2 Compiling

The STAR interpreter source code is partitioned into a set of 14 source files, as follows:

| | | | |
|---|---|---|---|
| stardefs.h, | starini1.c, | starini5.c, | starutil.c, |
| starcomm.h, | starini2.c, | starini6.c, | starlink.c. |
| starcode.c, | starini3.c, | starplus.c, | |
| starbifs.c, | starini4.c, | starhack.c, | |

All of these files except "starlink.c" may be compiled once and for all as STAR is installed on the host computer. If a single person is to be using STAR, it is useful to set up a special directory which contains object files for the STAR interpreter code, additional source code for "starlink.c", and both source and object files for all external function definitions. If several people are to be using STAR, it is perhaps better to compile the STAR source files minus "starlink.c", link the generated object files into a single object file and place this object file in a directory accessible to the various users. In this case, each user sets up a directory containing source and object code for "starlink.c" only, plus source and object code for his external function definitions.

When new external functions are defined or modifications are made to existing external functions, the files containing these definitions are recompiled, along with "starlink.c" if modified to reflect these changes.

## 5.3 Linking

After all source files for an application system have been compiled, the generated object files must be linked into a single executable file. In UNIX, the "cc" command which compiles C source files may also be used to link object files, possibly including sets of library functions. In VMS, the "LINK" command performs this function.

If the linking step is to be performed fairly often, it may be helpful to set up a command file containing a call to the linking mechanism along with specifications of the files to be linked. In this manner, it is only necessary to call the command file from the operating system level to invoke the linking process.

The generated executable file may be named as desired by the programmer. This file, when called from the operating system level, provides the STAR interpreter enhanced with the definitions of the specified external functions.

## 5.4 Loading STAR Definitions

Normally as an application system is being developed, the definitions for its symbolic structures are saved in the form of text on files using the built-in function "save". In this manner, if changes are made to any external functions, it is possible to recompile the relevant source files, relink the STAR interpreter and then reload the symbolic definitions from file in a fairly straightforward process.

If there are several text files which describe the symbolic structures of an application system and it is expected that the interpreter will require recompilation and relinking fairly often, it is simplest to set up a STAR user-defined function which calls "load" for each of the text files to be loaded. The definition of this function may be stored on a separate text file. When the symbolic structures for the application are to be loaded into STAR, it is then only necessary to load the one file containing this user-defined function and then to call the function. This process may be accomplished in a single step by using the STAR command-line arguments, as for example

```
star + loader.x : loadfunction
```

where "loader.x" is the file to be loaded and "loadfunction" is the user-defined function to be called.

## 5.5 Suspending STAR

When part or all of an application system appears to be in a fairly stable state, the built-in function "suspend" may be called to created a new executable file containing the STAR interpreter, associated external functions and definitions for symbolic structures. The single argument to "suspend" specifies the name of the new executable file to be created. Following this, the new executable file may be called from the operating system level in place of the original executable file for the STAR intepreter. If changes are to be made to external functions or symbolic structures contained within this new executable file, however, it will have to be discarded and a new version created reflecting the modifications. Usually, the "suspend" function is used only toward the end of the development phase and specifically to save the final version of the application system as a single entity.

## 5.6 Running the System

Given an executable file which contains the STAR interpreter, definitions for external functions and definitions for the symbolic structures of an application system, it is most desirable to be able to invoke the application system directly from the operating system level. This is possible if the entry into the application system is made to be a user-defined function taking no arguments. The call from the operating system then involves a command of the following form:

```
application : begin
```

where "application" is the name of the executable file containing the final version of STAR for the application system, and "begin" is the name of the user-defined function which initializes and executes the application system.

## 6. The STAR Utilities

This section describes the STAR utility functions, a set of compiled functions defined in C which may be called from external functions linked to the STAR interpreter. The utility functions allow external functions to extract data values from within UNIT structures, create new UNIT structures and modify existing UNIT structures.

In the description of each utility function, the arguments and result types are listed using the following codes.

UNIT - pointer to a UNIT structure,
INTEGER - single-precision fixed-point value,
REAL - single-precision floating-point value,
DOUBLE - double-precision floating-point value,
CHAR* - character string terminated by byte value 0.

For the UNIT type, it is recommended that a data type "unit" equivalent to "integer" be defined if possible within the source files for external functions (see Section 5.1). For the remaining types, the corresponding type as defined in the compiled language should be used (e.g., "DOUBLE" corresponds to "double" in C, "double precision" in FORTRAN and "real" in PASCAL). The exact types may vary slightly depending upon the particular compiler.

Also for each utility function, names for both the "by value" and "by reference" versions are provided. In their proper context of use, both versions of a utility may be treated as if their arguments and result values are of the indicated types. Due to constraints imposed by some FORTRAN compilers, the names for the "by reference" utilities are shortened to three letters per word with no contained underscore characters.

Additional information on the STAR utility functions appears as documentation within the source file "starutil.c".

### 6.1 Simple Utilities

#### Utilities involving UNITs in general

**"get_unit_type"** (by value)
**"getunityp"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "uni1": UNIT | INTEGER |

Returns an integer code indicating the UNIT type for "uni1".

1: NUMBER,
2: TOKEN,
3: STRING,
4: LIST,
5: RECORD,
6: EXPRESSION,
7: CONNECTION.

**"get_unit_usage_count"** (by value)
**"getuniusacou"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "uni1": UNIT | INTEGER |

Returns a nonnegative integer value indicating how many UNITs in STAR are currently employing the given UNIT as an element. If the count is greater than 1 and the UNIT is not a named RECORD, it is unwise to make direct modifications to the UNIT, as these may be reflected in unwanted alterations of the same UNIT as referenced elsewhere within the STAR knowledge base. To copy a UNIT, the utility "copy_unit" is used. For NUMBERs, TOKENs, STRINGs and CONNECTIONs, "copy_unit" forms a complete second UNIT as an image of the original. For LISTs, RECORDs and EXPRESSIONs, a new UNIT is formed at the top level only, containing references to the identical elements of the original UNIT.

Usage counts for named RECORDs are not critical in this sense, as only a single copy exists for each named RECORD and all changes are intended to apply in a global manner.

**"copy_unit"** (by value)
**"copuni"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "uni1": UNIT | UNIT |

If UNIT1 is a NUMBER, TOKEN, STRING or CONNECTION, an exact copy of the UNIT is formed and returned. If UNIT1 is a LIST, RECORD or EXPRESSION, a copy of the top level of the UNIT is formed, containing as elements the identical elements of the original UNIT.

This utility is often used following a call to "get_unit_count", which determines the number of other UNITs in the STAR knowledge base which contain a given UNIT as an element. When sharing is indicated by "get_unit_count", "copy_unit" may be used to form a private copy of the UNIT.

Note that it is possible to copy both unnamed and named RECORDs. Since STAR only recognizes a single RECORD of any given name, however, it is generally unwise to copy a named RECORD unless the name is to be removed in a subsequent operation.

For LISTs, RECORDs or EXPRESSIONs to be altered not only at the outer level, but within elements as well, "copy_unit" may be used individually upon the elements to be modified before these operations are performed. In this manner, it is possible to incrementally copy to the correct level of detail any UNIT to be modified.

**"get_unit_protection_code"** (by value)
**"getuniprocod"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "uni1": UNIT | INTEGER |

Returns an integer value "1" if "uni1" is part of the fixed portion of the STAR knowledge base, and "0" if not. In contrast to the STAR built-in functions, the STAR utilities may modify all UNIT data structures including fixed language structures. Such operations are performed at the risk of generating inconsistencies in the STAR knowledge base, however, and should only be done with full knowledge of the consequences. If the value "1" is returned, then it is recommended that the UNIT in question not be altered.

### Utilities involving NUMBERs

**"get_number"** (by value)
**"getnum"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "num1": UNIT | DOUBLE |

Extracts the numerical value from a STAR NUMBER. Note that this is a double precision floating point value (due to constraints imposed by C).

**"make_number"** (by value)
**"maknum"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "f": REAL | UNIT |

Creates a UNIT of type NUMBER with value "f". Returns a pointer to the new UNIT.

### Utilities involving TOKENs

**"get_token"** (by value)
**"gettok"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "tok1": UNIT | CHAR* |

Extracts the character string identification for a STAR TOKEN. Returns this string as the result.

**"make_token"** (by value)
**"maktok"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "s": CHAR* | UNIT |

Creates a UNIT of type TOKEN using the characters of string "s" as a guide. Returns a pointer to the new UNIT.

### Utilities involving STRINGs

**"get_string"** (by value)
**"getstr"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "str1": UNIT | CHAR* |

Extracts the character string within a STAR STRING. Returns this string as the result.

**"make_string"** (by value)
**"makstr"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "s": CHAR* | UNIT |

Creates a UNIT of type STRING using the characters of string "s" as a guide. Returns a pointer to the new UNIT.

### Utilities involving LISTs

**"get_list_size"** (by value)
**"getlissiz"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "lis1": UNIT | INTEGER |

Returns the number of elements in a LIST.

**"get_list_element"** (by value)
**"getlisele"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "lis1": UNIT | UNIT |
| "i": INTEGER | |

Extracts the "i"'th element of the specified LIST. Returns a pointer to this UNIT as the result.

**"make_list"** (by value)
**"maklis"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| (no arguments) | UNIT |

Creates a UNIT of type LIST containing zero elements. Returns a pointer to the new UNIT.

**"insert_list_at_head"** (by value)
**"inslisathea"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "lis1": UNIT | UNIT |
| "uni1": UNIT | |

Inserts the UNIT "uni1" as the new first element of the LIST "lis1". Returns "lis1" as the result.

**"insert_list_at_tail"** (by value)
**"inslisattai"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "lis1": UNIT | UNIT |
| "uni1": UNIT | |

Inserts the UNIT "uni1" at the end of the LIST "lis1". Returns "lis1" as the result.

### Utilities involving RECORDs

**"get_record_size"** (by value)
**"getrecsiz"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | INTEGER |

Returns the number of elements in a RECORD.

**"get_record_attribute"** (by value)
**"getrecatt"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "i": INTEGER | |

Extracts the "i"'th attribute from the specified RECORD. Returns a pointer to this UNIT as the result.

**"get_record_value"** (by value)
**"getrecval"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "i": INTEGER | |

Extracts the "i"'th value from the specified RECORD. Returns a pointer to this UNIT as the result.

**"match_record_value"** (by value)
**"matrecval"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "att1": UNIT | |

Extracts the value in "rec1" associated with the attribute "att1". Returns this UNIT as the result, or 0 if no value is found for the attribute.

**"make_record"** (by value)
**"makrec"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| (no arguments) | UNIT |

Creates a UNIT of type RECORD containing zero attribute-value pairs. Returns a pointer to the new UNIT.

**"get_named_record"** (by value)
**"getnamrec"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "s": CHAR* | UNIT |

Locates an existing named RECORD or creates a new named RECORD. The value "s" is a character string in the syntax of a STAR reference name. A pointer to the resulting RECORD is returned.

**"insert_record_at_head"** (by value)
**"insrecathea"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "att1": UNIT | |
| "uni1": UNIT | |

Inserts the attribute-value pair "att1" and "uni1" as the new first entry in the RECORD "rec1". Returns "rec1" as the result.

**"insert_record_at_tail"** (by value)
**"insrecattai"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "att1": UNIT | |
| "uni1": UNIT | |

Inserts the attribute-value pair "att1" and "uni1" at the end of the RECORD "rec1". Returns "rec1" as the result.

### Utilities involving EXPRESSIONs

**"get_expression_size"** (by value)
**"getexpsiz"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "exp1": UNIT | INTEGER |

Returns the number of elements (function + number of arguments) in an EXPRESSION.

**"get_expression_operation"** (by value)
**"getexpope"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "exp1": UNIT | UNIT |

Extracts the function being applied in an EXPRESSION. Returns a pointer to this named RECORD as the result.

**"get_expression_argument"** (by value)
**"getexparg"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "exp1": UNIT | UNIT |
| "i": INTEGER | |

Extracts the "i"'th argument from the specified EXPRESSION. Returns a pointer to this UNIT as the result.

**"make_expression"** (by value)
**"makexp"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |

Creates a UNIT of type EXPRESSION with the function "rec1" applied to zero arguments (the arguments may be entered singly using "insert_expression_at_tail"). Returns a pointer to the new EXPRESSION.

**"insert_expression_at_head"** (by value)
**"insexpathea"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "exp1": UNIT | UNIT |
| "uni1": UNIT | |

Inserts the UNIT "uni1" as the new first argument of the EXPRESSION "exp1". Returns "exp1" as the result.

**"insert_expression_at_tail"** (by value)
**"insexpattai"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "exp1": UNIT | UNIT |
| "uni1": UNIT | |

Inserts the UNIT "uni1" at the end of the EXPRESSION "exp1". Returns "exp1" as the result.

## Utilities involving CONNECTIONs

**"get_connection_contents"** (by value)
**"getconcon"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "con1": UNIT | INTEGER |

Extracts the contained data from a STAR CONNECTION. Returns this value as the result. STAR views the value as an integer, but depending on the actual value, it may be a constant or a pointer to an arbitrary data structure or function. It is up to the calling function to determine the actual type of the value.

In addition to "get_connection_contents", there are nine redundant copies which perform the same operation. These utilities are named "get_connection_contents_1" through "get_connection_contents_9" ("by reference" versions "getconcon1" through "getconcon9"). Each version of "get_connection_contents" may be declared within an external language as returning a value of a different type. This works around the conflict between strong data typing in some languages and the storage of arbitrarily-typed data values within STAR CONNECTIONs.

**"get_connection_label"** (by value)
**"getconlab"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "con1": UNIT | CHAR* |

Extracts the character string identification for a STAR CONNECTION. Returns this string as the result.

**"make_connection"** (by value)
**"makcon"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "d": INTEGER | UNIT |
| "t": CHAR* | |

This function creates a UNIT of type CONNECTION, given a data value of arbitrary type to store as the contents of the CONNECTION. Note that the value of "d" is only cast as an integer and may in fact be a more complicated value such as a pointer to a data structure or function.

The second argument is a character string to use as a pattern for the label of the CONNECTION. The pattern string is translated as follows. Any contained "at" characters ("@") in the pattern are converted to an ASCII representation of the data value (usually an address for a data structure or routine) to be placed in the "contents" field of the CONNECTION. Any lowercase letters in the pattern are converted to uppercase. Uppercase letters and all digits are left as they are and all other characters are converted to the underscore ("_") character. Thus, the final label for the CONNECTION contains only capital letters, digits and instances of the underscore character.

Examples of character strings which might be used for describing the labels of CONNECTIONs are the following:

C_ARRAY_OF_NUMBERS, GRAPH_@, PASCAL_SORT_FUNCTION.

The "@" character in the pattern "GRAPH_@", above, would be converted to an ASCII representation of the data value stored in the CONNECTION.

In addition to "make_connection", there are nine redundant copies named "make_connection_1" through "make_connection_9" ("by reference" versions "makcon1" through "makcon9"). These copies may be used in languages which require exact data type declarations to be made on the arguments of external functions. Each version of "make_connection" may then be declared as taking its first argument of a different type.

**"reassign_connection"** (by value)
**"reacon"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "con1": UNIT | UNIT |
| "d": INTEGER | |

Directly modify the CONNECTION "con1", substituting a new data value specified by "d". This utility is useful when it is desired to alter, without copying, a CONNECTION which is referenced in several places within the STAR knowledge base. Alternative use of the "make_connection" utility would form a separate CONNECTION, not acceptable in such cases.

In addition to "reassign_connection", there are nine redundant copies named "reassign_connection_1" through "reassign_connection_9" ("by reference" versions "reacon1" through "reacon9"). These copies may be used in languages which require exact data type declarations to be made on the arguments of external functions. Each version of "reassign_connection" may then be declared as taking its first argument of a different type.

**"relabel_connection"** (by value)
**"relcon"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "con1": UNIT | UNIT |
| "t": CHAR* | |

Directly modify the CONNECTION "con1", substituting a new label specified by "t". This utility is useful in similar cases to those requiring the use of "reassign_connection", where it is not sufficient to form a new copy of the CONNECTION with the desired label.

## 6.2 Incremental Utilities

### Incremental utilities involving LISTs

**"begin_list_scan"** (by value)
**"beglissca"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "lis1": UNIT | UNIT |
| "n": INTEGER | |

Starts a scanning operation through the LIST "lis1". Uses the integer "n" to reference this particular scan for this LIST, as more than one scan may be conducted simultaneously on the same LIST. Following a call to "begin_list_scan", the functions "get_current_list_element", "get_next_list_element", "insert_current_list_element", "insert_next_list_element" and "remove_current_list_element" may be used to incrementally retrieve values from the LIST and incrementally insert or remove values from the LIST. When no more operations are required for a particular scan, the function "end_list_scan" should be called.

The current position of scan "n" within LIST "lis1" is kept in an internal table. This table is updated as the scan progresses from head to tail in the LIST. When "end_list_scan" is called, the entry for "lis1" and "n" in this table is removed.

It is possible for a scan to reference a point which is prior to the beginning of a LIST or past the end of a LIST. Immediately following a call to "begin_list_scan", a call to "get_current_list_element" will return the pointer "0". To access the first element of the LIST at this point, it is necessary to use the function "get_next_list_element". Likewise, as a scan proceeds past the last element in a LIST, attempts to fetch the current element result in a return value of "0". In this case, it is still possible to use the function "insert_current_list_element", however, to insert a new element at the end of the LIST.

**"end_list_scan"** (by value)
**"endlissca"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "lis1": UNIT | UNIT |
| "n": INTEGER | |

Removes the entry for "lis1" and "n" from the internal table for LIST scanning operations. Returns the pointer to "lis1" as the result.

**"get_current_list_element"** (by value)
**"getcurlisele"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "lis1": UNIT | UNIT |
| "n": INTEGER | |

Returns the current element of "lis1" for scan number "n". There are four conditions under which a value of "0" is returned: (1) if "begin_list_scan" was not called for "lis1" and "n", (2) if the scan indicates a point prior to the first element of the LIST, (3) if the scan indicates a point following the end of the LIST, and (4) if the current element has been removed by the function "remove_current_list_element". Otherwise a pointer to a UNIT structure is returned.

**"get_next_list_element"** (by value)
**"getnexlisele"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "lis1": UNIT | UNIT |
| "n": INTEGER | |

Increments scan "n" on "lis1" to point to the next element of the LIST (this is the first element of the LIST if "begin_list_scan" was just called). Works also if the current element has been removed by "remove_current_list_element". Returns the new current element of "lis1" for scan "n". If scan "n" has passed the end of "lis1" or if

"begin_list_scan" was not called for the pair "lis1" and "n"; returns the 0 pointer.

**"insert_current_list_element"** (by value)
**"inscurlisele"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "lis1": UNIT | UNIT |
| "n": INTEGER | |
| "uni1": UNIT | |

Inserts "uni1" into "lis1" directly before the current element referenced by scan number "n" for "lis1". Works if scan "n" for "lis1" has passed the end of the LIST, in which case the new element is inserted at the end. Also works if the current element has been removed by "remove_current_list_element". Does not work, however, if scan "n" for "lis1" indicates the point prior to the first element. In this case, "insert_next_list_element" must be used. The UNIT inserted becomes the new current element, with the old current element (if any) directly following it. The entire "lis1" is returned as the result.

**"insert_next_list_element"** (by value)
**"insnexlisele"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "lis1": UNIT | UNIT |
| "n": INTEGER | |
| "uni1": UNIT | |

Inserts "uni1" into "lis1" directly after the current element referenced by scan number "n" for "lis1". Works if scan "n" for "lis1" indicates the point prior to the first element as well. In this case, the inserted UNIT becomes the new first element. Does not work, however, if scan "n" for "lis1" has passed the end of the LIST, in which case the function "insert_current_list_element" must be used. If scan "n" for "lis1" has passed the end of the LIST, or if "begin_list_scan" was not called for "lis1" and "n", returns "0". Otherwise, the UNIT inserted becomes the new "next" element, with the current element, if any, remaining as it was (even if previously removed by "remove_current_list_element"). The entire "lis1" is returned as the result.

**"remove_current_list_element"** (by value)
**"remcurlisele"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "lis1": UNIT | UNIT |
| "n": INTEGER | |

Removes the element of "lis1" currently referenced by scan "n" for "lis1". This results in a condition where there is no longer a current element which may be retrieved by "get_current_list_element", yet it is still possible to use the functions "get_next_list_element", "insert_current_list_element" and "insert_next_list_element".

Affects other scans for the same LIST only if they indicate the same current element. In this case, the other scans result as well in a condition where there is no current element.

Does not work if scan "n" for "lis1" indicates a point prior to the first element of the LIST or past the end of the LIST. The modified "lis1" is returned.

## Incremental utilities involving RECORDs

**"begin_record_scan"** (by value)
**"begrecsca"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "n": INTEGER | |

Starts a scanning operation through the RECORD "rec1". Uses the integer "n" to reference this particular scan for this RECORD, as more than one scan may be conducted simultaneously on the same RECORD. Following a call to "begin_record_scan", the functions "get_current_record_attribute", "get_current_record_value", "get_next_record_attribute", "get_next_record_value", "insert_current_record_entry", "insert_next_record_entry" and "remove_current_record_entry" may be used to incrementally retrieve values from the RECORD and incrementally insert or remove values from the RECORD. When no more operations are required for a particular scan, the function "end_record_scan" should be called.

The current position of scan "n" within RECORD "rec1" is kept in an internal table. This table is updated as the scan progresses from head to tail in the RECORD. When "end_record_scan" is called, the entry for "rec1" and "n" in this table is removed.

It is possible for a scan to reference a point which is prior to the beginning of a RECORD or past the end of a RECORD. Immediately following a call to "begin_record_scan", a call to "get_current_record_attribute" or "get_current_record_value" will return the pointer "0". To access the first entry of the RECORD at this point, it is necessary to use the function "get_next_record_attribute".

Likewise, as a scan proceeds past the last entry in a RECORD, attempts to fetch the current attribute or value result in a return value of "0". In this case, it is still possible to use the function "insert_current_record_entry", however, to insert a new entry at the end of the RECORD.

**"end_record_scan"** (by value)
**"endrecsca"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "n": INTEGER | |

Removes the entry for "rec1" and "n" from the internal table for RECORD scanning operations. Returns the pointer to "rec1" as the result.

**"get_current_record_attribute"** (by value)
**"getcurrecatt"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "n": INTEGER | |

Returns the attribute of the current RECORD entry for "rec1" and scan number "n". There are four conditions under which a value of "0" is returned: (1) if "begin_record_scan" was not called for "rec1" and "n", (2) if the scan indicates a point prior to the first entry of the RECORD, (3) if the scan indicates a point following the end of the RECORD, and (4) if the current entry has been removed by the function "remove_current_record_entry". Otherwise a pointer to a UNIT structure is returned.

**"get_current_record_value"** (by value)
**"getcurrecval"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "n": INTEGER | |

Returns the value of the current RECORD entry for "rec1" and scan number "n". There are four conditions under which a value of "0" is returned: (1) if "begin_record_scan" was not called for "rec1" and "n", (2) if the scan indicates a point prior to the first entry of the RECORD, (3) if the scan indicates a point following the end of the RECORD, and (4) if the current entry has been removed by the function "remove_current_record_entry". Otherwise a pointer to a UNIT structure is returned.

**"get_next_record_attribute"** (by value)
**"getnexrecatt"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "n": INTEGER | |

Increments scan "n" on "rec1" to point to the next entry of the RECORD (this is the first entry of the RECORD if "begin_record_scan" was just called). Works also if the current entry has been removed by "remove_current_record_entry". Returns the new current attribute of "rec1" for scan "n". If scan "n" has passed the end of "rec1" or if "begin_record_scan" was not called for the pair "rec1" and "n", returns the 0 pointer.

**"insert_current_record_entry"** (by value)
**"inscurrecent"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "n": INTEGER | |
| "att1": UNIT | |
| "uni1": UNIT | |

Inserts the attribute-value pair "att1" and "uni1" into "rec1" directly before the current entry referenced by scan number "n" for "rec1". Works if scan "n" for "rec1" has passed the end of the RECORD, in which case the new entry is inserted at the end. Also works if the current entry has been removed by "remove_current_record_entry". Does not work, however, if scan "n" for "rec1" indicates the point prior to the first entry. In this case, "insert_next_record_entry" must be used. The attribute-value pair inserted becomes the new current entry, with the old current entry (if any) directly following it. The entire "rec1" is returned as the result.

**"insert_next_record_entry"** (by value)
**"insnexrecent"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "n": INTEGER | |
| "att1": UNIT | |
| "uni1": UNIT | |

Inserts the attribute-value pair "att1" and "uni1" into "rec1" directly after the current entry referenced by scan number "n" for "rec1". Works if scan "n" for "rec1" indicates the point prior to the first entry as well. In this case, the inserted attribute and value become the new first entry. Does not work, however, if scan "n" for "rec1" has

passed the end of the RECORD, in which case the function "insert_current_record_entry" must be used. If scan "n" for "rec1" has passed the end of the RECORD, or if "begin_record_scan" was not called for "rec1" and "n", returns "0". Otherwise, the attribute-value pair inserted becomes the new "next" entry, with the current entry, if any, remaining as it was (even if previously removed by "remove_current_record_entry"). The entire "rec1" is returned as the result.

**"remove_current_record_entry"** (by value)
**"remcurrecent"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "rec1": UNIT | UNIT |
| "n": INTEGER | |

Removes the entry of "rec1" currently referenced by scan "n" for "rec1". This results in a condition where there is no longer a current entry which may be accessed by "get_current_record_attribute" or "get_current_record_value", yet it is still possible to use the functions "get_next_record_attribute", "insert_current_record_entry" and "insert_next_record_entry".

Affects other scans for the same RECORD only if they indicate the same current entry. In this case, the other scans result as well in a condition where there is no current element.

Does not work if scan "n" for "rec1" indicates a point prior to the first entry of the RECORD or past the end of the RECORD. The modified "rec1" is returned.

## Incremental utilities involving EXPRESSIONs

**"begin_expression_scan"** (by value)
**"begexpsca"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "exp1": UNIT | UNIT |
| "n": INTEGER | |

Starts a scanning operation through the EXPRESSION "exp1". Uses the integer "n" to reference this particular scan for this EXPRESSION, as more than one scan may be conducted simultaneously on the same EXPRESSION. Following a call to "begin_expression_scan", the functions "get_current_expression_argument", "get_next_expression_argument", "insert_current_expression_argum", "insert_next_expression_argument" and "remove_current_expression_argum" may be used to incrementally retrieve values from the EXPRESSION and incrementally insert or remove values from the EXPRESSION. When no more operations are required for a particular scan, the function "end_expression_scan" should be called.

The current position of scan "n" within EXPRESSION "exp1" is kept in an internal table. This table is updated as the scan progresses from head to tail in the EXPRESSION. When "end_expression_scan" is called, the entry for "exp1" and "n" in this table is removed.

It is possible for a scan to reference a point which is prior to the beginning of an EXPRESSION or past the end of an EXPRESSION. Immediately following a call to "begin_expression_scan", a call to "get_current_expression_argument" will return the pointer "0". To access the first argument of the EXPRESSION at this point, it is necessary to use the function "get_next_expression_argument". Likewise, as a scan proceeds past the last argument in an EXPRESSION, attempts to fetch the current argument result in a return value of "0". In this case, it is still possible to use the function "insert_current_expression_argum", however, to insert a new argument at the end of the EXPRESSION.

**"end_expression_scan"** (by value)
**"endexpsca"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "exp1": UNIT | UNIT |
| "n": INTEGER | |

Removes the entry for "exp1" and "n" from the internal table for EXPRESSION scanning operations. Returns the pointer to "exp1" as the result.

**"get_current_expression_argument"** (by value)
**"getcurexparg"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "exp1": UNIT | UNIT |
| "n": INTEGER | |

Returns the current argument of "exp1" for scan number "n". There are four conditions under which a value of "0" is returned: (1) if "begin_expression_scan" was not called for "exp1" and "n", (2) if the scan indicates a point prior to the first argument of the EXPRESSION, (3) if the scan indicates a point following the end of the EXPRESSION, and (4) if the current argument has been removed by the function "remove_current_expression_argum". Otherwise a pointer to a UNIT structure is returned.

**"get_next_expression_argument"** (by value)
**"getnexexparg"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "exp1": UNIT | UNIT |
| "n": INTEGER | |

Increments scan "n" on "exp1" to point to the next argument of the EXPRESSION (this is the first argument of the EXPRESSION if "begin_expression_scan" was just called). Works also if the current argument has been removed by "remove_current_expression_argum". Returns the new current argument of "exp1" for scan "n". If scan "n" has passed the end of "exp1" or if "begin_expression_scan" was not called for the pair "exp1" and "n", returns the 0 pointer.

**"insert_current_expression_argum"** (by value)
**"inscurexparg"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "exp1": UNIT | UNIT |
| "n": INTEGER | |
| "uni1": UNIT | |

Inserts "uni1" into "exp1" directly before the current argument referenced by scan number "n" for "exp1". Works if scan "n" for "exp1" has passed the end of the EXPRESSION, in which case the new argument is inserted at the end. Also works if the current element has been removed by "remove_current_expression_argum". Does not work, however, if scan "n" for "exp1" indicates the point prior to the first argument. In this case, "insert_next_expression_argument" must be used. The UNIT inserted becomes the new current argument, with the old current argument (if any) directly following it. The entire "exp1" is returned as the result.

**"insert_next_expression_argument"** (by value)
**"insnexexparg"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "exp1": UNIT | UNIT |
| "n": INTEGER | |
| "uni1": UNIT | |

Inserts "uni1" into "exp1" directly after the current argument referenced by scan number "n" for "exp1". Works if scan "n" for "exp1" indicates the point prior to the first argument as well. In this case, the inserted UNIT becomes the new first argument. Does not work, however, if scan "n" for "exp1" has passed the end of the EXPRESSION, in which case the function "insert_current_expression_argum" must be used. If scan "n" for

"expl" has passed the end of the EXPRESSION, or if "begin_expression_scan" was not called for "expl" and "n", returns "0". Otherwise, the UNIT inserted becomes the new "next" argument, with the current argument, if any, remaining as it was (even if previously removed by "remove_current_expression_argum"). The entire "expl" is returned as the result.

**"remove_current_expression_argum"** (by value)
**"remcurexparg"** (by reference)

| ARGUMENTS: | RESULT: |
|---|---|
| "expl": UNIT | UNIT |
| "n": INTEGER | |

Removes the argument of "expl" currently referenced by scan "n" for "expl". This results in a condition where there is no longer a current argument which may be retrieved by "get_current_expression_argument", yet it is still possible to use the functions "get_next_expression_argument", "insert_current_expression_argum" and "insert_next_expression_argument".

Affects other scans for the same EXPRESSION only if they indicate the same current argument. In this case, the other scans result as well in a condition where there is no current argument.

Does not work if scan "n" for "expl" indicates a point prior to the first argument of the EXPRESSION or past the end of the EXPRESSION. The modified "expl" is returned.

## Appendix: Summary of Built-in Structures

The following listing describes the names and organizational categories of all built-in named RECORDs in STAR. On-line information may be obtained by typing in the reference names for individual named RECORDs inside the STAR environment.

The entries for built-in functions, below, indicate data types for arguments and results. The conventions used in specifying the types are listed in the beginning of Section 3.3 of the Reference Manual. An exception to these conventions is that the arguments to the functions are not numbered in the following listing. Abbreviation symbols for the built-in functions are also listed in parentheses immediately following the reference names.

### I. Built-in Classes

concept

class
attribute
function
variable
element
rule

boolean
rule_mode

### II. Built-in Attributes

#### 1. For all named RECORDs

name
member_of

comment

#### 2. For classes

subclass_of
members
subclasses

pattern
aspects

value
default
if_needed
if_asserted
if_retracted

### 3. For attributes

side_effects

### 4. For functions

abbreviation
n_arguments

arguments
temporary

algorithm

### 5. For variables

bindings

### 6. For rules

mode

condition
action

### 7. For general programming

case

result_value
break_value
skip_value
return_value
stop_value

## III. Built-in Functions

### 1. For NUMBERs

| | | |
|---|---|---|
| negate(-) | (NUMBER) | -> NUMBER |
| add(+) | (NUMBER NUMBER) | -> NUMBER |

| | | |
|---|---|---|
| subtract(-) | (NUMBER NUMBER) | -> NUMBER |
| multiply(*) | (NUMBER NUMBER) | -> NUMBER |
| divide(/) | (NUMBER NUMBER) | -> NUMBER |
| | | |
| minimum | (NUMBER NUMBER) | -> NUMBER |
| maximum | (NUMBER NUMBER) | -> NUMBER |

**2. For TOKENs**

**(as names)**

| | | |
|---|---|---|
| locate | (TOKEN) | -> <concept> |
| test | (TOKEN) | -> <concept> |

**3. For STRINGs**

| | | |
|---|---|---|
| character | (STRING NUMBER) | -> STRING |
| fetch | (STRING NUMBER) | -> STRING |
| release | (STRING NUMBER) | -> STRING |
| join | (STRING STRING) | -> STRING |
| find | (STRING STRING) | -> NUMBER |
| length | (STRING) | -> NUMBER |

**4. For LISTs**

**(as lists)**

| | | |
|---|---|---|
| select | (LIST NUMBER) | -> UNIT |
| replace | (LIST NUMBER UNIT) | -> LIST |
| delete | (LIST NUMBER) | -> LIST |
| insert | (LIST NUMBER UNIT) | -> LIST |
| take | (LIST NUMBER) | -> LIST |
| drop | (LIST NUMBER) | -> LIST |
| append | (LIST LIST) | -> LIST |
| size | (LIST) | -> NUMBER |

**(as sets)**

| | | |
|---|---|---|
| union | (LIST LIST) | -> LIST |
| intersection | (LIST LIST) | -> LIST |
| difference | (LIST LIST) | -> LIST |

**5. For RECORDs**

**(unnamed)**

| | | |
|---|---|---|
| get | (RECORD <attribute>) | -> UNIT |

| | | |
|---|---|---|
| put | (RECORD <attribute> UNIT) | -> RECORD |
| omit | (RECORD <attribute>) | -> RECORD |
| detach | (RECORD LIST) | -> RECORD |
| attach | (RECORD RECORD) | -> RECORD |
| key | (RECORD) | -> LIST |
| image | (RECORD) | -> LIST |
| build | (LIST LIST) | -> RECORD |

**(named)**

| | | |
|---|---|---|
| define | (RECORD) | -> <concept> |
| create | (<concept> <class>) | -> <concept> |
| assert | (<concept> <attribute> UNIT) | -> <concept> |
| retract | (<concept> <attribute>) | -> <concept> |
| modify | (<concept> <attribute> UNIT) | -> <concept> |
| revise | (<concept> LIST) | -> <concept> |
| merge | (<concept> RECORD) | -> <concept> |

**(variables)**

| | | |
|---|---|---|
| dot(.) | (<variable>) | -> UNIT |
| new | (<variable> UNIT) | -> <variable> |
| set | (<variable> UNIT) | -> <variable> |
| old | (<variable>) | -> <variable> |

**(classified)**

| | | |
|---|---|---|
| determine | (<concept> <attribute>) | -> UNIT |
| estimate | (<concept> <attribute>) | -> UNIT |
| calculate | (<concept> <attribute>) | -> UNIT |
| obtain | (<concept> <attribute> <attribute>) | -> UNIT |
| path | (<concept>) | -> LIST |

**(classes)**

| | | |
|---|---|---|
| enumerate(:) | (<class>) | -> LIST |

**6. For EXPRESSIONs**

| | | |
|---|---|---|
| operation | ('EXPRESSION) | -> <function> |
| application | ('EXPRESSION) | -> LIST |
| formulate | (<function> LIST) | -> EXPRESSION |

## 7. Logic functions

### (predicates)

| | | |
|---|---|---|
| number | (UNIT) | -> <boolean> |
| token | (UNIT) | -> <boolean> |
| string | (UNIT) | -> <boolean> |
| list | (UNIT) | -> <boolean> |
| record | (UNIT) | -> <boolean> |
| expression | (UNIT) | -> <boolean> |
| connection | (UNIT) | -> <boolean> |
| null | (UNIT) | -> <boolean> |

### (relations)

| | | |
|---|---|---|
| equal(=) | (UNIT UNIT) | -> <boolean> |
| less(<) | (NUMBER NUMBER) | -> <boolean> |
| greater(>) | (NUMBER NUMBER) | -> <boolean> |
| in | (UNIT LIST) | -> <boolean> |
| subset | (LIST LIST) | -> <boolean> |
| isa | (<concept> <class>) | -> <boolean> |
| within | (<class> <class>) | -> <boolean> |

### (boolean)

| | | |
|---|---|---|
| not(~) | (<boolean>) | -> <boolean> |
| and(&) | (<boolean> <boolean>) | -> <boolean> |
| or(\|) | (<boolean> <boolean>) | -> <boolean> |

### (quantifiers)

| | | |
|---|---|---|
| exists | (LIST <variable> 'UNIT) | -> <boolean> |
| every | (LIST <variable> 'UNIT) | -> <boolean> |
| which | (LIST <variable> 'UNIT) | -> LIST |

## 8. Input/Output functions

### (terminal)

| | | |
|---|---|---|
| parse | () | -> UNIT |
| display | (UNIT) | -> UNIT |
| input | () | -> STRING |
| output | (STRING) | -> STRING |

| | | |
|---|---|---|
| format | (STRING LIST) | -> STRING |

**(file)**

| | | |
|---|---|---|
| save | (STRING) | -> STRING |
| stash | (STRING LIST) | -> STRING |
| load | (STRING) | -> STRING |
| read | (STRING) | -> STRING |
| write | (STRING STRING) | -> STRING |
| extend | (STRING STRING) | -> STRING |

**(string)**

| | | |
|---|---|---|
| spell | (UNIT) | -> STRING |
| unspell | (STRING) | -> UNIT |
| scan | (STRING) | -> UNIT |

## 9. Programming functions

**(evaluation)**

| | | |
|---|---|---|
| quote(') | (UNIT) | -> UNIT |
| evaluate | ('UNIT) | -> UNIT |
| prepare | (UNIT) | -> UNIT |
| apply | (<function> LIST) | -> UNIT |

**(conditional)**

| | | |
|---|---|---|
| if | (<boolean> 'UNIT) | -> UNIT |
| ifelse | (<boolean> 'UNIT 'UNIT) | -> UNIT |
| branch | (UNIT LIST) | -> UNIT |

**(compound evaluation)**

| | | |
|---|---|---|
| do | (LIST) | -> UNIT |
| repeat | (LIST) | -> UNIT |
| while | ('UNIT LIST) | -> UNIT |
| for | ('UNIT 'UNIT 'UNIT LIST) | -> UNIT |
| through | (LIST <variable> LIST) | -> UNIT |

**(rule-based operation)**

| | | |
|---|---|---|
| invoke | (<class> LIST) | -> UNIT |

**(control flow)**

| | | |
|---|---|---|
| result | (UNIT) | -> RECORD |
| break | (UNIT) | -> RECORD |
| skip | () | -> RECORD |
| return | (UNIT) | -> RECORD |
| stop | (UNIT) | -> RECORD |

### 10. Miscellaneous functions

| | | |
|---|---|---|
| pause | (STRING) | -> STRING |
| system | (STRING) | -> STRING |
| suspend | (STRING) | -> STRING |
| exit | () | -> ... |

## IV. Built-in Variables

pound_sign

control
alternatives

## V. Built-in Elements

### 1. Boolean values

true
false

### 2. Rule modes

single_test
single_application
multiple_application

### 3. Miscellaneous

nil

TECHNICAL REPORT STANDARD TITLE PAGE

| 1. Report No. 85-89 | 2. Government Accession No. | 3. Recipient's Catalog No. |
|---|---|---|
| 4. Title and Subtitle<br>STAR (Simple Tool for Automated Reasoning) | | 5. Report Date<br>December 15, 1985<br>6. Performing Organization Code |
| 7. Author(s)<br>Gary C. Borchardt | | 8. Performing Organization Report No. |
| 9. Performing Organization Name and Address<br>JET PROPULSION LABORATORY<br>California Institute of Technology<br>4800 Oak Grove Drive<br>Pasadena, California 91109 55574450 | | 10. Work Unit No.<br>11. Contract or Grant No.<br>NAS7-918<br>13. Type of Report and Period Covered<br>External Report<br>JPL Publication |
| 12. Sponsoring Agency Name and Address<br>NATIONAL AERONAUTICS AND SPACE ADMINISTRATION<br>Washington, D.C. 20546 | | 14. Sponsoring Agency Code<br>RE156 BK-506-45-11-01-00 |

15. Supplementary Notes

16. Abstract

STAR is an interactive, interpreted programming language for the development and operation of Artificial Intelligence application systems. The language is intended for use primarily in the development of software application systems which rely on a combination of symbolic processing, central to the vast majority of AI algorithms, with routines and data structures defined in complied languages such as C, FORTRAN, and PASCAL. References to routines and data structures defined in complied languages are intermixed with symbolic structures in STAR, resulting in a hybrid operating environment in which symbolic and non-symbolic processing and organization of data may interact to a high degree within the execution of particular application systems.

The STAR language was developed in the course of a project involving AI techniques in the interpretation of imaging spectrometer data and is derived in part from a previous language called CLIP. The interpreter for STAR is implemented as a program defined in the language C and has been made available for distribution in source code form through NASA's Computer Software Management and Information Center (COSMIC). Contained within this report are the STAR Tutorial Guide, which introduces the language in a step-by-step manner, and the STAR Reference Manual, which provides a detailed summary of the features of STAR.

| 17. Key Words (Selected by Author(s)) | 18. Distribution Statement |
|---|---|
| Computer Programming and Software;<br>Cybernetics | Unclassified; unlimited |

| 19. Security Classif. (of this report) | 20. Security Classif. (of this page) | 21. No. of Pages | 22. Price |
|---|---|---|---|
| Unclassified | Unclassified | | |